UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

PROYECTO FIN DE CARRERA

Análisis, Diseño e Implementación de un módulo Linux para el acceso a servidores NFS

AUTOR: Carlos Rodríguez Miguel

TUTOR: Félix García Carballeira Octubre, 2015

Agradecimientos

En primer lugar agradecer a mi familia, muy especialmente a mis padres, por haberme dado todos los medios, el apoyo y la motivación que a veces me faltaba para permitirme llegar a donde he llegado. Gracias de todo corazón por ser como sois.

En segundo lugar quiero agradecer a los compañeros de “fatigas” universitarias, con los que tan buenos momentos he pasado y seguiremos pasando. Gracias a todas aquellas personas con las que he trabajado, haciendo prácticas y memorias interminables.

Gracias sinceramente a todos mis amigos y a todas las personas que de verdad se alegran por mí y por este día.

Aprovecho también para agradecer a todo el equipo humano que conforman los diferentes integrantes de todos y cada uno de los departamentos de la universidad, siempre dispuestos a colaborar aportándome entre tanto infinidad de valores durante mi estancia en la misma.

Por último, y como no podía ser de otro modo, a mi tutor Félix, por toda la ayuda y comprensión mostrada, sin ti este proyecto no hubiera sido posible.

Contenido

Agradecimientos 2
1 Introducción 8
1.1 Objetivos del proyecto 8
1.2 Antecedentes 10
1.3 Estructura del Documento 11
2 Módulos del núcleo 13
2.1 El sistema operativo 13
2.1.1 Introducción 13
2.1.2 El núcleo 14
2.1.3 Tipos de núcleos 16
2.2 Sistema operativo Linux 21
2.2.1 El núcleo del sistema operativo Linux 21
2.2.2 Espacio de usuario y espacio de kernel 22
2.3 Los módulos del núcleo de Linux 24
2.3.1 Administración de módulos 25
2.3.2 Carga de módulos 27
2.3.3 Descarga de módulos 29
2.4 Drivers 31
2.4.1 Tipos de drivers 32
3 Llamadas a procedimientos remotos (RPC) 35
3.1 Mecanismos de comunicación en sistemas distribuidos 35
3.2 Remote procedure call (RPC) 36
3.2.1 Historia 36
3.2.2 Introducción 36
3.2.3 Terminología 38
3.2.4 El modelo de RPC 40
3.2.5 Protocolo de transporte 41
3.2.6 Localización del servidor 42
3.2.7 Modelo genérico de comunicación 43
3.2.8 Semántica de Fallos 44
3.3 RPC de SUN 45
3.3.1 Programación con RPC 46

3.3.2 Requisitos del protocolo RPC de SUN 46
3.3.3 Programas y procedimientos 47
3.3.4 Autenticación 48
3.3.5 La asignación del número de programa 48
3.3.6 Otros usos del protocolo RPC 49
3.3.7 Protocolo del programa PortMapper 49
3.4 Lenguaje RPC de SUN 52
3.4.1 Definiciones 52
3.4.2 Estructuras 53
3.4.3 Uniones 53
3.4.4 Enumerados 54
3.4.5 Tipos 54
3.4.6 Constantes 55
3.4.7 Programas 55
3.4.8 Declaraciones 56
3.4.9 Casos especiales 58
3.5 Ejemplo de RPC 59
3.6 Entornos de objetos distribuidos 63
4 Sistemas de ficheros distribuidos 64
4.1 Introducción 64
4.2 Diseño 65
4.3 Terminología 65
4.4 Componentes de un SFD 66
4.4.1 Servicios de un SFD 66
4.4.2 Servicio de directorios 67
4.4.3 Servicio de ficheros 70
4.5 Fiabilidad 77
4.6 Replicación 77
5 Sistema de ficheros en red NFS 79
5.1 Introducción 79
5.2 Definición del protocolo NFS 80
5.2.1 Modelo del sistema de ficheros 80
5.2.2 Autenticación en RPC 81

5.2.3 Constantes del servicio 82
5.2.4 Tamaños de las estructuras de XDR 82
5.2.5 Tipos de datos básicos 83
5.2.6 Procedimientos del servidor 89
5.3 Problemas de implementación de NFS 96
5.3.1 Relación entre Servidor / Cliente 96
5.3.2 Interpretación del pathname 96
5.3.3 Problemas de permisos 97
5.4 Definición del protocolo de montado 97
5.4.1 Introducción 97
5.4.2 Información adicional del protocolo 98
5.4.3 Constantes del servicio 98
5.4.4 Tamaño de las estructuras XDR 98
5.4.5 Tipo de datos básicos 98
5.4.6 Procedimientos del servidor 99
6 Desarrollo de módulos del kernel de Linux 103
6.1 Introducción 103
6.2 Implementación de un módulo cargable (LKM) 104
6.3 Asignación dinámica de números mayores 105
6.4 Makefile para los módulos del núcleo 106
6.5 Exportar símbolos 109
6.6 Multiproceso simétrico 109
6.7 Comunicación con los procesos 110
6.7.1 Ficheros especiales de dispositivos 110
6.7.2 File_operations 111
6.7.3 Número mayor y menor del dispositivo 114
6.7.4 El sistema de ficheros /proc 115
6.8 Uso de RPC desde un módulo 115
6.8.1 Protocolo del programa PortMapper 115
6.9 El programa calcular 118
6.9.1 Constantes del servicio 118
6.9.2 Tipos de datos 119
6.9.3 Procedimientos del servidor 119

6.9.4 Funciones XDR ... 120
6.10 Interfaz de usuario ... 121
6.10.1 Proceso en modo usuario ... 121
7 Diseño e implementación del módulo cliente de NFS ... 122
7.1 Interfaz de usuario ... 122
7.1.1 Protocolo mount ... 124
7.1.2 Protocolo NFS ... 125
7.2 Acceso al módulo mediante ioctl ... 128
7.3 Implementación de las funciones ... 131
7.3.1 Portmapper ... 132
7.3.2 Protocolo mount ... 133
7.3.3 Protocolo NFS ... 137
7.4 Funciones XDR ... 151
7.4.1 Tipos de datos básicos ... 151
7.4.2 Protocolo mount ... 152
7.4.3 Protocolo NFS ... 153
8 Evaluación ... 156
8.1 Entorno de las pruebas ... 156
8.2 Definición de las pruebas ... 156
8.3 Propósito de las pruebas ... 156
8.4 Resultados ... 157
8.4.1 Creación y borrado ... 157
8.4.2 Lectura ... 158
8.4.3 Lectura y escritura (copiar) ... 160
8.5 Resumen ... 163
9 Presupuesto del proyecto ... 164
9.1 Costes del personal ... 164
9.2 Coste Hardware y licencias ... 165
9.3 Coste material fungible ... 166
9.4 Coste oficina ... 166
9.5 Coste final del proyecto ... 167
10 Conclusiones ... 168
10.1 Trabajos futuros ... 168

11 Bibliografía 169

1 Introducción

1.1 Objetivos del proyecto

El objetivo general del proyecto es el diseño e implementación de un módulo cargable de Linux que actúa como cliente de NFS. Se exponen los pasos necesarios para desarrollar un módulo del sistema operativo Linux cargable y descargable en tiempo de ejecución. El software ha sido desarrollado en lenguaje C y actúa de manera similar a los servicios de un micronúcleo, pero con la diferencia de que se ejecutan en el espacio de memoria del núcleo. El optar por un sistema de ficheros en red no fue una decisión arbitraria, el hito perseguido fue adquirir el conocimiento a bajo nivel de NFS.

Para llegar al mentado objetivo comencé realizando una exhaustiva tarea de investigación en lo que se refiere a la creación de un módulo cargable del kernel, durante la misma procedí a realizar un módulo simple del más que conocido “Hola Mundo”.

Una vez realizado el primer módulo cargable se analizaron las diferentes opciones existentes para poder dotar de funcionalidad al módulo, lo que me llevo a implementar un módulo encargado de realizar operaciones aritméticas en red, en concreto la suma de dos parámetros.

Para llevarlo a cabo fue necesario permitir la comunicación entre procesos. Principalmente hay dos formas de que un módulo del núcleo se comunique con los procesos, a través de los controladores de dispositivos:

- Ficheros especiales de dispositivos, */dev*:
  - Dispositivo especial de caracteres.
  - Dispositivo especial de bloques.
- Sistema de ficheros, */proc*.

En el desarrollo del presente proyecto se optó por usar los ficheros especiales de dispositivos, en concreto un dispositivo especial de caracteres, para lo que fue necesario definir la estructura de operaciones permitidas para el dispositivo. Dado que los ficheros de dispositivos se supone que representan dispositivos físicos, y la mayoría de los dispositivos físicos se utilizan para salida y para entrada, se deben definir todas aquellas funciones que no lo estén ya en *file_operations* (estructura de operaciones) implementándose con la función especial IOCTL, input output control.

Uno de los principales motivos por el cual programar en el núcleo, es dar soporte a algún tipo de hardware facilitando a los programas su acceso mediante un interfaz de comunicación. En nuestro caso en concreto se registra un controlador para el dispositivo especial de caracteres *“ficticio”*, no va a estar ligado a ningún hardware.

Cómo el objetivo final del proyecto era implementar un módulo cargable en el kernel de Linux que actúe como cliente de NFS, sistema de ficheros en red, se utilizaron las RPC´s de SUN como sistema de comunicación remota.

En el kernel no hay ningún un precompilador de RPC, como rpcgen, que nos genera automáticamente el código fijo y relacionado con la parte de comunicaciones, *stubs* o suplentes. De modo que se deben definir los distintos programas o servicios que va a utilizar. Por tanto tuve que desarrollar un conjunto de llamadas de RPC, *Remote Procedure Call*, y XDR, *eXternal Data Representation*, estas últimas son necesarias para poder realizar con éxito la comunicación entre máquinas que pueden ejecutar en arquitecturas diferentes.

Para permitir que un módulo del kernel realice llamadas RPC a un servidor debe implementar el servicio portmap, el cual permite registrar servicios en el servidor asociándolos a un puerto por el que escuchará peticiones, desregistra estos servicios así como permite su consulta.

La interfaz de usuario esta implementada en lenguaje C que se ejecuta en modo usuario y que permite realizar las llamadas *ioctl* a través del dispositivo especial de caracteres para hacer llegar las peticiones al módulo del núcleo.

Finalmente, con las ideas más claras, con la experiencia adquirida en la creación de un módulo cargable en el kernel de Linux con comunicación entre procesos y que permitía realizar operaciones en red, se realizó el análisis y diseño del módulo de Linux cargable para el acceso a servidores NFS.

Fue necesario realizar un profundo estudio del protocolo NFS y MOUNT para llevar a cabo el diseño y la implementación del módulo cliente, basándome en las especificaciones dadas por dichos protocolos.

NFS proporciona un acceso transparente a los sistemas de ficheros compartidos sobre redes de área local, está diseñado para ser independiente del sistema operativo, la arquitectura de red o del protocolo de transporte utilizado. Esta independencia se obtiene mediante el uso RPC, se utilizó la versión 2 del protocolo para la realización del proyecto y por tanto fue necesario implementar cada una de las funciones del servidor MOUNT para su posterior uso por el cliente NFS.

El protocolo de montado permite que el servidor controle el acceso remoto a un conjunto de clientes. Algunos sistemas operativos proporcionan una operación de montado para hacer que todos los sistemas de ficheros aparezcan como uno solo árbol de directorios, mientras que otros mantienen el árbol de sistemas de ficheros, de manera que existan varios sistemas de ficheros en uno. NFS busca un componente con una ruta en un momento dado.

Tras su implementación procedí a realizar una evaluación del mismo comparando los tiempos de ejecución de la solución implementada frente al NFS que proporciona el sistema operativo.

Por lo tanto podemos decir que entre los objetivos del proyecto se encontraban:

- Conocer el desarrollo de módulos del kernel.
- Conocer el funcionamiento de NFS.
- Desarrollar un módulo del kernel de Linux que se comporte como cliente de NFS.
- Realizar una evaluación del mismo.

1.2 Antecedentes

Cuando comencé a realiza el proyecto fin de carrera tenía los conocimientos de programación en modo usuario del lenguaje C, adquiridos durante mi formación universitaria, así como del uso de RPC´s en este mismo modo. También poseía conocimientos de NFS y Linux. No obstante me enfrentaba al reto de programar a bajo nivel de forma que el software desarrollado se integrará en el sistema operativo, por lo que el código del mismo debía ser especialmente robusto.

Linux es el kernel de un sistema operativo de código abierto basado en Unix y es uno de los ejemplos de software libre más conocidos.

Se distribuye bajo licencia GPL v2 y actualmente es desarrollado por colaboradores de todo el mundo. Linux, inicialmente, fue desarrollado por Linus Torvalds, en 1991, y al ser de software libre, permitió que desarrolladores y usuarios de todo el mundo adaptaran códigos de otros proyectos para su uso en este nuevo sistema.

Actualmente el núcleo recibe contribuciones de miles de programadores y es normalmente empaquetado junto a distintos tipos de software y distribuido a través de una "distribución de Linux".

El núcleo de Linux está organizado siguiendo una arquitectura monolítica, en la cual, todas las partes del núcleo del sistema operativo (sistemas de ficheros, manejadores de dispositivos, protocolos de red, etc.) están enlazadas como una sola imagen (normalmente el fichero /vmlinuz) que es la que se carga y ejecuta en el arranque del sistema.

Esta estructura podría dar lugar a un sistema poco flexible, ya que cualquier funcionalidad que se le quisiera añadir al núcleo del sistema requeriría una recompilación completa del mismo. Aun así, la filosofía de fuentes abiertos hace Linux mucho más flexible que otros sistemas operativos en la que los fuentes no están disponibles.

Esta limitación desapareció con la incorporación, en la versión 2.0 de Linux, del soporte para la carga dinámica de módulos en el núcleo. Esta nueva característica permite la incorporación en caliente de nuevo código al núcleo del sistema operativo, sin necesidad de reinicializar el sistema.

El proyecto aborda el desarrollo de un pequeño módulo, calcular, que servirá de introducción y facilitará la compresión del módulo que implementa un cliente NFS.

1.3 Estructura del Documento

El presente documento queda estructurado como sigue.

En el primer capítulo, al cual pertenece este apartado, se realiza una introducción en la que se exponen cuáles eran los objetivos principales del proyecto y se narra cómo partiendo de mis conocimientos iniciales sobre el tema, conseguí llegar a ellos.

En el segundo capítulo se abordan los módulos del núcleo de Linux, comenzaremos realizando una pequeña introducción de los distintos tipos de sistemas operativos existentes para terminar profundizando en el sistema operativo utilizado en la realización del proyecto, Linux.

Se realiza una introducción tanto a nivel del sistema operativo Linux, como a nivel de núcleo para describir finalmente los módulos del núcleo de Linux, así como los tipos de drivers.

El tercer capítulo define las llamadas a procedimientos remotos, las cuales permiten la comunicación de procesos en un sistema distribuido.

Se realiza una introducción y se exponen los mecanismos de comunicación en los sistemas distribuidos para abordar en profundidad el mecanismo utilizado en la realización del proyecto, las RPC.

El cuarto capítulo tiene por objetivo definir qué es un sistema de ficheros distribuido, SFD.

Se realiza una pequeña introducción y se muestra una arquitectura típica del diseño de un sistema de ficheros distribuido, para finalmente profundizar en los componentes necesarios en todo sistema de ficheros distribuidos y los servicios que proporcionan, así como ahondar en ciertas características como la fiabilidad y la replicación.

El quinto capítulo se encarga de describir los servicios de los protocolos NFS y MOUNT, las constantes del servicio, el tamaño de las estructuras XDR, los tipos de datos básicos así como los procedimientos del servidor.

Comienza realizando una introducción al sistema de ficheros en red de SUN, NFS, y abarca desde su definición en la que se exponen las diferentes versiones existentes, para centrarnos en la versión 2 que se ha utilizado para la implementación del proyecto, pasando por el modelo del sistema, la autenticación en RPC y los problemas de implementación.

El objetivo del sexto capítulo es describir, de la forma más exhaustiva posible, el proceso completo de implementación de un módulo cargable del kernel, el uso de las RPC´s desde el mismo así como el sistema de comunicación entre los procesos.

Se explicara en detalle un ejemplo sencillo que describe la implementación de un módulo de Linux que actúa como cliente del programa calcular.

El séptimo capítulo detalla el diseño y la implementación del módulo de Linux para el acceso a servidores NFS versión 2.

Este capítulo expone la interfaz de usuario implementada tanto para el protocolo NFS como para el MOUNT, el acceso al módulo a través del dispositivo especial de caracteres, la implementación de las llamadas ioctl y de las operaciones del portmapper en los protocolos MOUNT y NFS así como las funciones y los tipos de datos XDR desarrollados para ambos protocolos.

En el octavo capítulo se realiza una evaluación en tiempos de ejecución entre el sistema implementado y el servicio NFS que facilita el sistema operativo Linux.

El noveno capítulo esta destina a la valoración del proyecto.

Siendo el décimo capítulo las conclusiones a las que se han llegado con la realización del presente proyecto, se incluyen dos apartados para indicar los posibles trabajos futuros a realizar así como los problemas encontrados durante la realización del mismo.

2 Módulos del núcleo

El objetivo del presente capitulo es explicar los módulos del núcleo de Linux, para ello necesitamos conocer que es un sistema operativo y en concreto cómo es el sistema sobre el que se va a desarrollar el proyecto, Linux. Posteriormente nos encaminamos a profundizar sobre el núcleo, pasando por los distintos tipos de núcleos existentes para comprender las características del núcleo sobre el que se desarrolla el proyecto.

Se establecen las diferencias entre el espacio de usuario y el del kernel, se explica cómo se realiza la carga y descarga de módulos en el núcleo, así como los distintos controladores de dispositivo existentes.

2.1 El sistema operativo

Un sistema operativo es un programa o conjunto de programas de un sistema informático que gestiona los recursos de hardware y provee servicios a los programas de aplicación, ejecutándose en modo privilegiado respecto de los restantes (aunque puede que parte de él se ejecute en espacio de usuario).

Uno de los propósitos del sistema operativo que gestiona el núcleo intermediario consiste en gestionar los recursos de localización y protección de acceso del hardware, hecho que alivia a los programadores de aplicaciones de tener que tratar con estos detalles. La mayoría de aparatos electrónicos que utilizan microprocesadores para funcionar, llevan incorporado un sistema operativo: teléfonos móviles, reproductores de DVD, computadoras, radios, enrutadores, etc.

2.1.1 Introducción

Un **sistema operativo** es la parte del **software** que se comunica directamente con el hardware, conociendo los detalles de bajo nivel que necesita para ello. Un sistema operativo puede ser de propósito general, como es Windows, Linux o Android, o bien puede ser de propósito específico, como puede ser el sistema operativo que usa un teléfono móvil para las telecomunicaciones, el de un coche o el de un satélite. Un dispositivo puede tener un sistema híbrido, contando con un sistema operativo de propósito general y otro de propósito específico, como sucede en los móviles.

Solo profundizaremos en los sistemas operativos de propósito general. Las utilidades y características de estos son las siguientes:

- **Abstrae del hardware**: Esto hace posible que los programas desarrollados para el sistema operativo en cuestión funcionen en distintos dispositivos, sin conocer sus detalles específicos. Es el sistema operativo el que se encarga de comunicarlo con el hardware.
- **Proporciona una biblioteca de métodos**: Estos métodos o funciones pueden ser usados por los programadores a la hora de desarrollar sus aplicaciones.
- **Gestiona los recursos de manera equitativa**: Un sistema operativo debe encargarse de que los procesos progresen en tiempo de ejecución. Un solo procesador debe encargarse de un gran número de procesos simultáneamente, y no es posible hacerlo en paralelo. El sistema operativo los gestiona para que todos los procesos avancen de manera ecuánime.

- **Debe consumir el mínimo de recursos**: Para que los procesos se realicen de la manera más rápida posible, el sistema operativo debe consumir los menos recursos posibles, de manera que la mayor parte de recursos se dedique a estos procesos.
- **Sirve de interfaz para el usuario**: Un sistema operativo puede implementar múltiples interfaces.

Figura 2.1 Interacción del sistema operativo

En ciertos textos, el sistema operativo es llamado indistintamente como núcleo o kernel, pero debe tenerse en cuenta que esta identidad entre kernel y sistema operativo es solo cierta si el núcleo es monolítico, un diseño común entre los primeros sistemas. En caso contrario, es incorrecto referirse al sistema operativo como núcleo.

2.1.2 El núcleo

No necesariamente se necesita un núcleo para usar una computadora. Los programas pueden cargarse y ejecutarse directamente en una computadora «vacía», siempre que sus autores quieran desarrollarlos sin usar ninguna abstracción del hardware ni ninguna ayuda del sistema operativo. Ésta era la forma normal de usar muchas de las primeras computadoras, para usar distintos programas se tenía que reiniciar y reconfigurar la computadora cada vez. Con el tiempo, se empezó a dejar en memoria (aún entre distintas ejecuciones) pequeños programas auxiliares, como el cargador y el depurador, o se cargaban desde memoria de sólo lectura. A medida que se fueron desarrollando, se convirtieron en los fundamentos de lo que llegarían a ser los primeros núcleos de sistema operativo.

El núcleo o kernel es la parte fundamental del sistema operativo, se trata de la capa visible del software más baja del sistema que provee y gestiona los recursos del sistema de forma segura

a través de las llamadas al sistema y se define como la parte que se ejecuta en modo privilegiado o modo núcleo.

Es el principal responsable de facilitar a los distintos programas acceso seguro al hardware de la computadora. Como hay muchos programas y el acceso al hardware es limitado, también se encarga de decidir qué programa podrá hacer uso de un dispositivo de hardware y durante cuánto tiempo, lo que se conoce como multiplexado. Acceder al hardware directamente puede ser realmente complejo, por lo que los núcleos suelen implementar una serie de abstracciones del hardware. Esto permite esconder la complejidad, y proporciona una interfaz limpia y uniforme al hardware subyacente, lo que facilita su uso al programador.

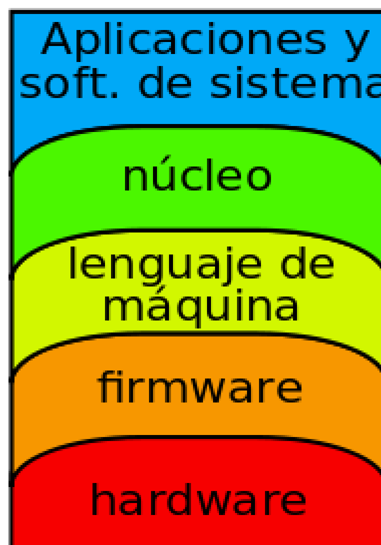

Figura 2.2 Arquitectura de computadores

Los núcleos tienen como funciones básicas garantizar la carga y la ejecución de los procesos, las entradas/salidas y proponer una interfaz entre el espacio núcleo y los programas del espacio del usuario.

Aparte de las funcionalidades básicas, el conjunto de las funciones de los puntos siguientes (incluidos los pilotos materiales, las funciones de redes y sistemas de ficheros o los servicios) necesariamente no son proporcionadas por un núcleo de sistema de explotación. Pueden establecerse estas funciones del sistema de explotación tanto en el espacio usuario como en el propio núcleo. Su implantación en el núcleo se hace con el único objetivo de mejorar los resultados. En efecto, según la concepción del núcleo, la misma función llamada desde el espacio usuario o el espacio núcleo tiene un coste temporal obviamente diferente. Si esta llamada de funciones es frecuente, puede resultar útil integrar estas funciones al núcleo para mejorar los resultados.

El núcleo de un sistema operativo suele operar en modo privilegiado. Al operar en dicho modo un error de programación en el núcleo del sistema operativo puede resultar en un error fatal del cual el sistema sólo puede recuperarse mediante el reinicio del sistema.

2.1.3 Tipos de núcleos

Los sistemas operativos se pueden clasificar en base a la cantidad de funcionalidad implementada en su núcleo. Hay cuatro grandes tipos de núcleos:

- Los **núcleos monolíticos** facilitan abstracciones del hardware subyacente realmente potentes y variadas.
- Los **micronúcleos** o microkernel proporcionan un pequeño conjunto de abstracciones simples del hardware, y usan las aplicaciones llamadas servidores para ofrecer mayor funcionalidad.
- Los **núcleos híbridos** (micronúcleos modificados) son muy parecidos a los micronúcleos puros, excepto porque incluyen código adicional en el espacio de núcleo para que se ejecute más rápidamente.
- Los **exonúcleos** no facilitan ninguna abstracción, pero permiten el uso de bibliotecas que proporcionan mayor funcionalidad gracias al acceso directo o casi directo al hardware.

2.1.3.1 Núcleo monolítico

Es una arquitectura de sistema operativo donde éste en su totalidad trabaja en espacio del núcleo, estando él solo en modo supervisor. Difiere de otras arquitecturas, como la de micronúcleo, en que solo define una interfaz virtual de alto nivel sobre el hardware del ordenador. Un conjunto primitivo de llamadas al sistema implementa todos los servicios propios del sistema operativo tales como la planificación de procesos, concurrencia, sistema de archivos, gestión de memoria, etc.

En esta arquitectura hay una correspondencia entre el programa que conforma el sistema operativo y el núcleo en sí.

Éste núcleo está programado de forma no modular y puede tener un tamaño considerable. A su vez, cada vez que se añada una nueva funcionalidad, deberá ser recompilado en su totalidad y luego reiniciado. Todos los componentes funcionales del núcleo tienen acceso a todas sus estructuras de datos internas y a sus rutinas. Por ende, un error en una rutina podría propagarse a todo el sistema.

Hay diversas ramificaciones de este diseño, que se han ido amoldando a nuevas necesidades. Existen sistemas que, en tiempo de ejecución, permiten la carga dinámica de módulos ejecutables, lo cual le brinda al modelo de núcleo monolítico algunas de las ventajas de un micronúcleo. Dichos módulos pueden ser compilados, modificados, cargados y descargados en tiempo de ejecución, de manera similar a los servicios de un micronúcleo, pero con la diferencia de que se ejecutan en el espacio de memoria del núcleo mismo (anillo 0). De esta forma, es probable que un bloqueo del módulo bloquee todo el núcleo. Además, el módulo pasa a formar un todo con el núcleo, usando la API del mismo, y no se emplea un sistema de mensajes como en los micronúcleos. Este es el esquema usado por, entre otros, Linux, FreeBSD y varios derivados de UNIX.

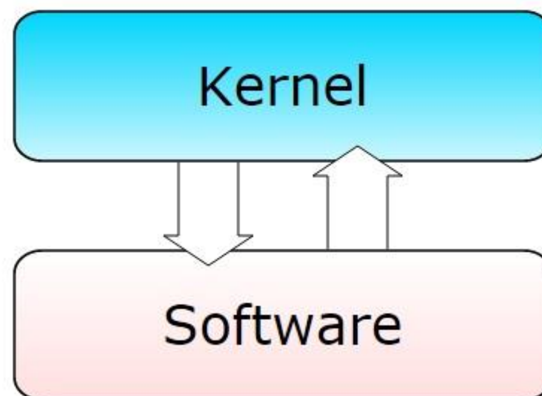

Figura 2.3 Núcleo monolítico

Entre los sistemas operativos que cuentan con núcleos monolíticos se encuentran:

- Núcleos tipo Unix
    - Linux
    - Syllable
    - Unix
        - BSD (FreeBSD, NetBSD, OpenBSD)
        - Solaris
- Núcleos tipo DOS
    - DR-DOS
    - MS-DOS
    - Familia Microsoft Windows 9x (95, 98, 98SE, Me)
- Núcleos del Mac OS hasta Mac OS 8.6
- OpenVMS
- XTS-400

2.1.3.2 Micronúcleo

Es un tipo de núcleo de un sistema operativo que provee un conjunto de primitivas o llamadas mínimas al sistema para implementar servicios básicos como espacios de direcciones, comunicación entre procesos y planificación básica. Todos los otros servicios (gestión de memoria, sistema de archivos, operaciones de E/S, etc.), que en general son provistos por el núcleo, se ejecutan como procesos servidores en espacio de usuario.

El paradigma del micronúcleo, tuvo una gran relevancia académica durante los años ochenta y principios de los noventa, dentro de lo que se denominó *self healing computing*, esto es, sistemas independientes que fuesen capaces de superar por si mismos errores de software o hardware. En un principio pretendía ser una solución a la creciente complejidad de los sistemas operativos.

Se basa en una programación altamente modular, y tiene un tamaño mucho menor que el núcleo monolítico. Como consecuencia, el refinamiento y el control de errores son más rápidos y sencillos. Además, la actualización de los servicios es más sencilla y ágil, ya que sólo es

necesaria la recompilación del servicio y no de todo el núcleo. Como contraprestación, el rendimiento se ve afectado negativamente.

Las principales ventajas de su utilización son la reducción de la complejidad, la descentralización de los fallos (un fallo en una parte del sistema no se propagaría al sistema entero) y la facilidad para crear y depurar controladores de dispositivos. Según los defensores de esta tendencia, esto mejora la tolerancia a fallos y eleva la portabilidad entre plataformas de hardware.

Por otro lado, sus principales dificultades son la complejidad en la sincronización de todos los módulos que componen el micronúcleo y su acceso a la memoria, la anulación de las ventajas de *Zero Copy* y la integración con las aplicaciones. Además, los procesadores y arquitecturas modernas de hardware están optimizados para sistemas de núcleo que pueden mapear toda la memoria. Sus detractores le achacan también y fundamentalmente, mayor complejidad en el código, menor rendimiento y limitaciones en diversas funciones.

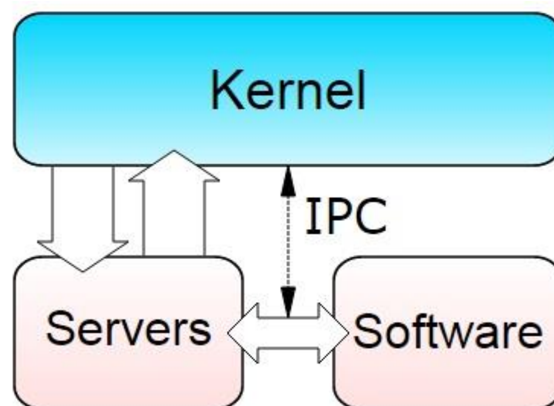

Figura 2.4 Micronúcleo

Entre los sistemas operativos con micronúcleo podemos citar:

- AIX
- Minix
- NeXTSTEP
- RaOS
- QNX
- SymbOS
- AmigaOS
- Hurd
- L4
- RadiOS
- SO3
- AmayaOS
- Amoeba
- MorphOS
- Netkernel
- ChorusOS
- Symbian

Nota: algunos consideran NeXTSTEP un núcleo híbrido.

2.1.3.3 Núcleo híbrido

Es un tipo de núcleo de un sistema operativo, básicamente, es un micronúcleo que tienen algo de código «no esencial» en espacio de núcleo, para que éste se ejecute más rápido de lo que lo haría si estuviera en espacio de usuario.

Éste fue un compromiso que muchos desarrolladores de los primeros sistemas operativos, con arquitectura basada en micronúcleo, adoptaron antes que se mostrara que los micronúcleos pueden tener también muy buen rendimiento.

La mayoría de sistemas operativos modernos pertenecen a esta categoría, siendo el más popular Microsoft Windows. El núcleo de Mac OS X, XNU, también es un micronúcleo NO modificado, debido a la inclusión de código del núcleo de *FreeBSD* en el núcleo basado en *Mach*. *DragonFlyBSD* es el primer sistema *BSD* que adopta una arquitectura de núcleo híbrido sin basarse en *Mach*.

Se tiende a confundir erróneamente a los núcleos híbridos con los núcleos monolíticos que pueden dinámicamente cargar módulos después del arranque. El concepto de núcleo híbrido se refiere a que el núcleo en cuestión usa mecanismos o conceptos de arquitectura tanto del diseño monolítico como del micronúcleo, específicamente el paso de mensajes y la migración de código «no esencial» hacia el espacio de usuario (manteniendo a su vez cierto código «no esencial» en el propio núcleo por razones de rendimiento).

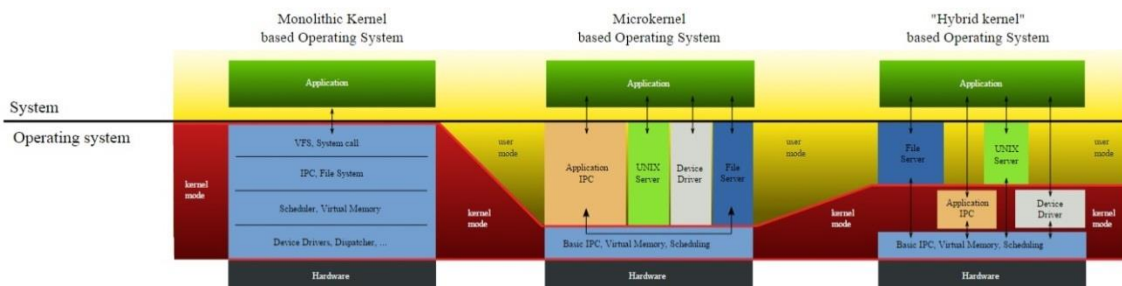

Figura 2.5 Estructura de núcleo monolítico, micronúcleo y núcleo híbrido

Entre los sistemas operativos con núcleo hibrido podemos citar:

- Microsoft Windows NT, usado en todos los sistemas que usan el código base de Windows NT
- XNU (usado en Mac OS X)
- DragonFlyBSD
- ReactOS

2.1.3.4 Exonúcleo

Los exonúcleos, también conocidos como sistemas operativos verticalmente estructurados, representan una aproximación radicalmente nueva al diseño de sistemas operativos.

La idea subyacente es permitir que el desarrollador tome todas las decisiones relativas al rendimiento del hardware. Los exonúcleos son extremadamente pequeños, ya que limitan expresamente su funcionalidad a la protección y el multiplexado de los recursos. Se llaman así porque toda la funcionalidad deja de estar residente en memoria y pasa a estar fuera, en bibliotecas dinámicas.

Los diseños de núcleos clásicos (tanto el monolítico como el micronúcleo) abstraen el hardware, escondiendo los recursos bajo una capa de abstracción del hardware, o detrás de los controladores de dispositivo. En los sistemas clásicos, si se asigna memoria física, nadie puede estar seguro de cuál es su localización real, por ejemplo.

La finalidad de un exonúcleo es permitir a una aplicación que solicite una región específica de la memoria, un bloque de disco concreto, etc., y simplemente asegurarse que los recursos pedidos están disponibles, y que el programa tiene derecho a acceder a ellos.

Debido a que el exonúcleo sólo proporciona una interfaz al hardware de muy bajo nivel, careciendo de todas las funcionalidades de alto nivel de otros sistemas operativos, éste es complementado por una «biblioteca de sistema operativo». Esta biblioteca se comunica con el exonúcleo subyacente, y facilita a los programadores de aplicaciones las funcionalidades que son comunes en otros sistemas operativos.

Algunas de las implicaciones teóricas de un sistema exonúcleo son que es posible tener distintos tipos de sistemas operativos (por ejemplo Windows, Unix) ejecutándose en un solo exonúcleo, y que los desarrolladores pueden elegir prescindir o incrementar funcionalidades por motivos de rendimiento.

Actualmente, los diseños exonúcleo están fundamentalmente en fase de estudio y no se usan en ningún sistema popular. Un concepto de sistema operativo es Némesis, creado por la Universidad de Cambridge, la Universidad de Glasgow, Citrix Systems y el Instituto Sueco de Informática. El MIT también ha diseñado algunos sistemas basados en exonúcleos. Los exonúcleos se manejan en diferente estructura dado que también cumplen funciones distintas

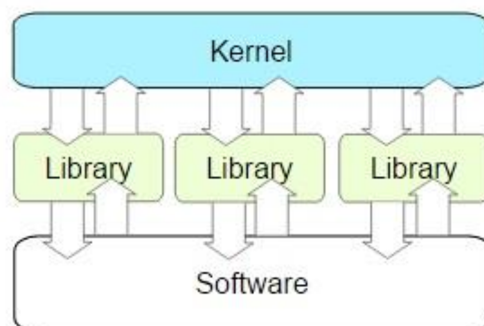

Figura 2.6 Exonúcleo

2.2 Sistema operativo Linux

Linux es un sistema operativo monolítico, es decir, es un único programa de gran tamaño donde todos los componentes funcionales del núcleo tienen acceso a todas sus estructuras de datos internas y a sus rutinas. Lo cual significa que todos los subsistemas funcionan en el mismo modo privilegiado y comparten el mismo espacio de direcciones, la comunicación entre ellos es realizada por medio de las llamadas usuales de funciones de C.

La mayoría de los sistemas operativos actuales comparten la característica de ser modulares, para permitir que algunas tareas que no puede hacer el kernel inicialmente, se logren después de agregar el módulo apropiado. Linux permite la carga de módulos por medio de la opción CONFIG_MODULES en tiempo de compilación.

Los sistemas operativos con estructura de micronúcleo poseen las partes funcionales del núcleo divididas en unidades separadas con mecanismos de comunicación estrictos entre ellos. Esto hace que la integración de nuevos componentes en el núcleo mediante el proceso de configuración tarde bastante tiempo.

Cita de Linus Torvalds:

“... el paso de mensajes como la operación fundamental del SO es sólo un ejercicio de masturbación de la ciencia de la computación. Quizás suene bien, pero actualmente no tienes nada HECHO.”

En un sistema Linux la interacción final con los dispositivos la realizan los controladores o el kernel. Dicho de otra forma, un dispositivo sólo podrá ser usado si el kernel lo soporta o si existe un controlador capaz de manejarlo y se configura apropiadamente para hacerlo. Si queremos usar un controlador de dispositivo para un dispositivo que no tengamos integrado en el núcleo, tendremos que configurar el núcleo para poder usar el dispositivo. La otra opción que tenemos es que Linux nos permite cargar y descargar componentes del sistema operativo dinámicamente según los vaya necesitando.

La ventaja de que sea modular, para el caso de Linux, es que se pueden agregar o descargar módulos sin necesidad de reiniciar el sistema y en algunos casos sin necesidad de recompilar el kernel. Las fuentes en C de cada versión del kernel cuentan con controladores para diversos dispositivos. Cuando se compila una versión, algunos de esos controladores pueden unirse con el kernel, estáticamente, otros pueden dejarse como módulos para cargarse / descargarse cuando la parte estática del kernel esté operando y otros pueden ser excluidos del proceso de compilación, y por lo tanto no podrán ser usados ni cuando el kernel esté operando.

2.2.1 El núcleo del sistema operativo Linux

El núcleo de Linux es un programa único y grande, enlazado a partir de sus numerosos módulos objeto constituyentes, empieza a ejecutarse poco después de arrancar el ordenador. Su principal función es esconder los detalles del hardware a los programas que sobre él se van a ejecutar. Es el encargado de que el software y el hardware de tu ordenador puedan trabajar juntos.

*El **kernel** o **núcleo** nos ofrece una interfaz simple con nuestro hardware*.

El kernel de Linux es **modular**, esto quiere decir, que está construido de tal forma que podemos añadirle "trozos" de kernel durante la marcha, y si ciertas partes del mismo no se están usando, podemos eliminarlas tranquilamente.

Cada kernel tiene su propia manera de manejar el hardware, y tiene sus propias llamadas al sistema definidas. Debe conocer el *"idioma"* de nuestro hardware, los **drivers** definen el protocolo de comunicación entre los dispositivos y el núcleo.

El núcleo del sistema es el responsable de ofrecer una máquina extendida y manejar los recursos (CPU, memoria y periféricos). Sus funciones son:

- Planificación del tiempo de CPU. Administración del tiempo de procesador que utilizan los programas en ejecución.
- Manejo de la memoria (reserva, protección e intercambios). Administración de la memoria para todos los programas en ejecución.
- Manejo de dispositivos (*drivers*). Es el encargado de que se pueda acceder a los periféricos, elementos del ordenador, de una manera cómoda.
- Seguridad de dispositivos, procesos y usuarios.
- Contabilidad de uso de CPU y espacio en disco.
- Comunicación entre procesos (memoria compartida, semáforos y mensajes).

"El kernel permite al desarrollador olvidarse de cómo se gestionan los recursos hardware y centrarse en resolver los problemas que se presentan a su aplicación."

El administrador debe conocer el núcleo, los aspectos básicos, cómo modificarlo y adaptarlo, para poder realizar una serie de operaciones. Entre ellas:

- Modificar el *hardware* (añadir dispositivos, procesadores, memoria, etc.).
- Optimizar el rendimiento en cuanto a velocidad y el aprovechamiento de la memoria y los dispositivos.

En Linux, el núcleo se sitúa en el directorio raíz o en */boot*. Suele llamarse *vmlinuz-sufijo*, donde el sufijo puede no existir o indicar el número de versión. Es un fichero comprimido producto de la compilación de una mezcla de código "C" y ensamblador. Este código fuente se puede "bajar" directamente en formato .tar.gz, por ejemplo, o bien instalarse como un paquete más.

2.2.2 Espacio de usuario y espacio de kernel

Para programar un módulo es importante distinguir entre el espacio de usuario, user space, y el espacio de kernel, kernel space.

- Espacio del kernel: El sistema operativo Linux y en especial su kernel se ocupan de gestionar los recursos de hardware de la máquina de una forma eficiente y sencilla, ofreciendo al usuario una interfaz de programación simple y uniforme. El kernel, y en especial sus drivers, constituyen así un puente o interface entre el programador de

aplicaciones para el usuario final y el hardware. Toda subrutina que forma parte del kernel tales como los módulos o drivers se consideran que están en el espacio del kernel, kernel space.

- Espacio de usuario: Los programas que utiliza el usuario final, tales como las shell u otras aplicaciones, residen en el espacio de usuario, user space. Como es lógico estas aplicaciones necesitan interaccionar con el hardware del sistema, pero no lo hacen directamente, sino a través de las funciones que soporta el kernel.

Esto lo podemos apreciar en la siguiente figura:

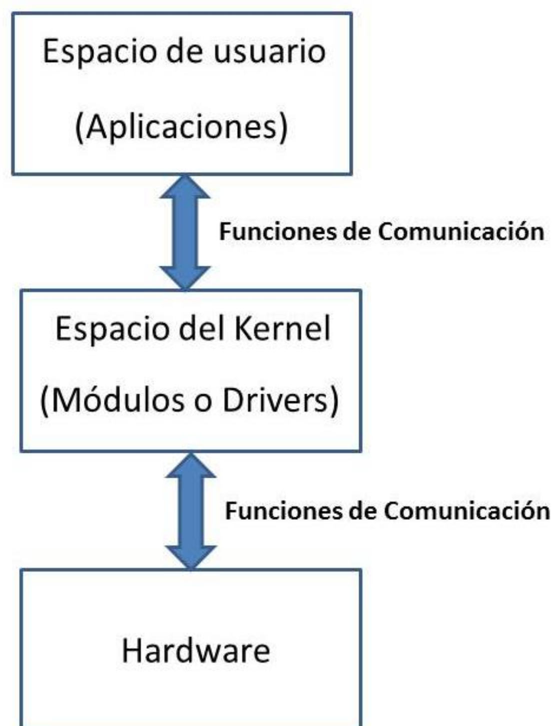

Figura 2.7 Espacio de usuario y espacio del kernel

Un módulo se ejecuta en espacio de kernel, mientras que cualquier aplicación que desarrollemos se ejecuta en espacio de usuario. La CPU tiene distintos niveles de ejecución, están comprendidos desde ring0 a ring3, cada uno con unos privilegios determinados. En los módulos Linux utiliza el ring0, modo superusuario, para los procesos que se van a ejecutar con privilegios de área de kernel mientras que ring3, modo usuario, será el utilizado para ejecutar las aplicaciones en área de usuario, de forma que el código de un programa pueda pasar de un nivel a otro. Se pasa de la ejecución de espacio de usuario a espacio de kernel a través de llamadas al sistema o a través de interrupciones para posteriormente poder realizar una determinada tarea del kernel o bien atender a una interrupción como puede ser la de un periférico, operaciones asíncronas, al contrario de cualquier operación sobre los procesos que se realizan de forma síncrona.

2.3 Los módulos del núcleo de Linux

El núcleo del sistema es un programa grande y complejo que funciona en el modo privilegiado del procesador, por lo cual puede leer y escribir en los puertos y tiene acceso a toda la memoria.

Dado que disponemos del código fuente del núcleo, la máxima flexibilidad imaginable en cuanto a sus características y configuración se alcanza modificando ese código y recompilándolo. Para evitar tener que recompilar cada vez que se cambia el *hardware* o se necesita añadir o eliminar alguna funcionalidad, algunas partes del código objeto no se enlazan con el resto del núcleo en tiempo de compilación: son los módulos.

Podemos entonces decir que un módulo es una pieza de código objeto que está especialmente preparado para integrarse dinámicamente con el resto del kernel.

Las implicaciones de esta definición son enormes, pues programar un módulo es programar una porción del kernel, crear una parte del kernel en sí mismo, implementando llamadas al sistema y manejando el hardware a bajo nivel.

Típicamente, pero no en todos los casos, se compilan como módulos los manejadores de dispositivo, device drivers.

Cuándo enlazar directamente en el núcleo:

1. Cuando el dispositivo o servicio se usa en el arranque (ide, scsi, etc.)
2. Cuando el dispositivo o servicio es de uso frecuente o constante (tarjeta de red, sistemas de ficheros, etc).

Cuándo compilar como módulo:

1. Cuando el dispositivo o servicio se usa de tarde en tarde (escáner, puerto paralelo, etc.)
2. Cuando el dispositivo puede variar sus características (dispositivo externo scsi, discos reemplazables en caliente, etc.)
3. Manejadores no incluidos en la versión oficial del núcleo (experimentales, locales, especiales, etc.)

La ventaja de la programación de módulos es la incrustación o embebido que tienen dentro del kernel, su carga es dinámica por lo que permite una carga y descarga rápida sin necesidad de estar dentro del kernel, esto es lo que se conoce como **modularización**. También son útiles para comprobar el nuevo código del núcleo sin tener que volver a crear el núcleo y reiniciar el ordenador cada vez que se compruebe. Pero tienen un inconveniente, hay un significante decremento en el rendimiento y en la memoria asociada con los módulos del núcleo. Un módulo cargable debe proveer un poco más de código, lo que unido a las estructuras de datos adicionales hace que un módulo ocupe un poco más de memoria. Hay también un nivel de “indirección introducido” que hace que los accesos de los recursos del núcleo sean bastante menos eficientes para los módulos.

Los **módulos** de Linux son trozos de código, controladores de dispositivos o servicio, que se pueden quedar vinculados dinámicamente en el núcleo en cualquier momento después de que el sistema haya arrancado. Es decir se pueden cargar cuando el superusuario o algún dispositivo lo solicitan. Una vez que ya no se necesitan, se pueden desvincular y quitar del núcleo. Mediante el modelo cliente-servidor. Los módulos se pueden compilar, carga y descargar en cualquier momento sin necesidad de recompilar todo el núcleo ni de reiniciar la computadora. Para dar esta funcionalidad, los módulos incorporan tres partes: Una de inicialización en la que se ejecutan las instrucciones necesarias al cargar el módulo, ***init_module()***, otra con las funciones que debe desempeñar el módulo y la última, de finalización, ***cleanup_module()***, con las funciones que se han de hacer cuando el módulo se descargue.

El núcleo está compuesto de módulos que en su mayoría son controladores de dispositivo, pseudo-controladores de dispositivo como controladores de red, o sistemas de archivos. Los módulos que se distribuyen con el kernel están ubicados en el directorio */lib/modules/version*, donde *version* es la versión del kernel, con la extensión "*.o*" organizados en directorios que indican el tipo de dispositivo o el propósito, por ejemplo: *fs* - sistema de archivos, *net* - protocolos y hardware para redes.

2.3.1 Administración de módulos

La administración de los módulos a nivel básico es sencilla, limitándose a los procesos de inserción, consulta de los instalados y extracción. Lógicamente, sólo el superusuario puede administrar directamente el núcleo y los módulos.

- Para saber qué módulos hay instalados se usa la orden **lsmod**. Lista los módulos que están actualmente cargados en el kernel, junto con algunos datos adicionales de interés sobre el estado de los mismos, tamaño, cuenta de usos y lista de módulos que lo usan.
- Para instalar un módulo en el kernel actualmente cargado o en ejecución se usa **insmod** o **modprobe**. Este último comprueba dependencias entre módulos e inserta todos los necesarios, no sólo el que le pasamos como argumento.
    - Insmod: Trata de cargar el módulo especificado. Pueden pasarse opciones específicas para el módulo, a continuación del nombre con la sintaxis símbolo = valor. Puede indicarse una ruta no estándar para buscar módulos estableciéndola en la variable MODPATH o en /etc/modules.conf. Dado que los módulos se enlazan directamente con el kernel, deben ser compilados para una versión precisa, con la opción -f puede evitarse el chequeo de versiones.
    - Depmod: Permite calcular las dependencias existentes entre los módulos, entre todos los disponibles con la opción –a, y las guarda en el fichero /lib/modules/$(uname -r)/modules.dep. Como ya he comentado, los módulos se encuentran siempre en directorios con el nombre de versión del núcleo, a partir de /lib/modules.
    - Modprobe: Emplea la información de dependencias generada por depmod e información de /etc/modules.conf para cargar el módulo especificado, cargando antes todos los módulos de los cuales dependa. Para especificar el módulo basta escribir el nombre (sin la ruta, ni la extensión .o) o uno de los alias definidos en

/etc/modutils/alias. Si hay líneas pre-install o post-install en /etc/modules.conf, modprobe puede ejecutar un comando antes y o después de cargar el módulo. Como opciones para cargar el módulo usa prioritariamente las dadas en la línea de comandos y después las especificadas en líneas de la forma options módulo opciones en el archivo /etc/modules.conf.

- Para extraer un módulo del kernel usamos la orden **rmmod**. Descarga uno o más módulos cargados, mientras estos no estén siendo usados. Con la opción -r intenta descargar recursivamente módulos de los cuales el módulo especificado dependa. El comando **rmmod -a** descarga todos los módulos que no estén siendo usados.

La inserción y extracción de módulos se encuentra normalmente automatizada mediante un hilo de ejecución del núcleo, ***kmod***, que llama a modprobe cuando un proceso le pide abrir un dispositivo que no conoce o un servicio del que no dispone. Por ejemplo:

- Cuando se pide arrancar un servicio de red sobre Ethernet y no existe el dispositivo instalado, kmod ejecuta modprobe eth0.
- Cuando se pide montar un disco scsi y no está el manejador de la tarjeta scsi instalado, kmod ejecuta modprobe scsi_hostadapter.
- Si un proceso intenta abrir un fichero especial de dispositivo de bloques con número mayor, major number, *M* y menor, minor number, *m*, kmod ejecuta modprobe block-major-M-n.

Modprobe hace uso del fichero */etc/modules.conf* que le permite traducir estos nombres simbólicos a nombres de módulos.

2.3.2 Carga de módulos

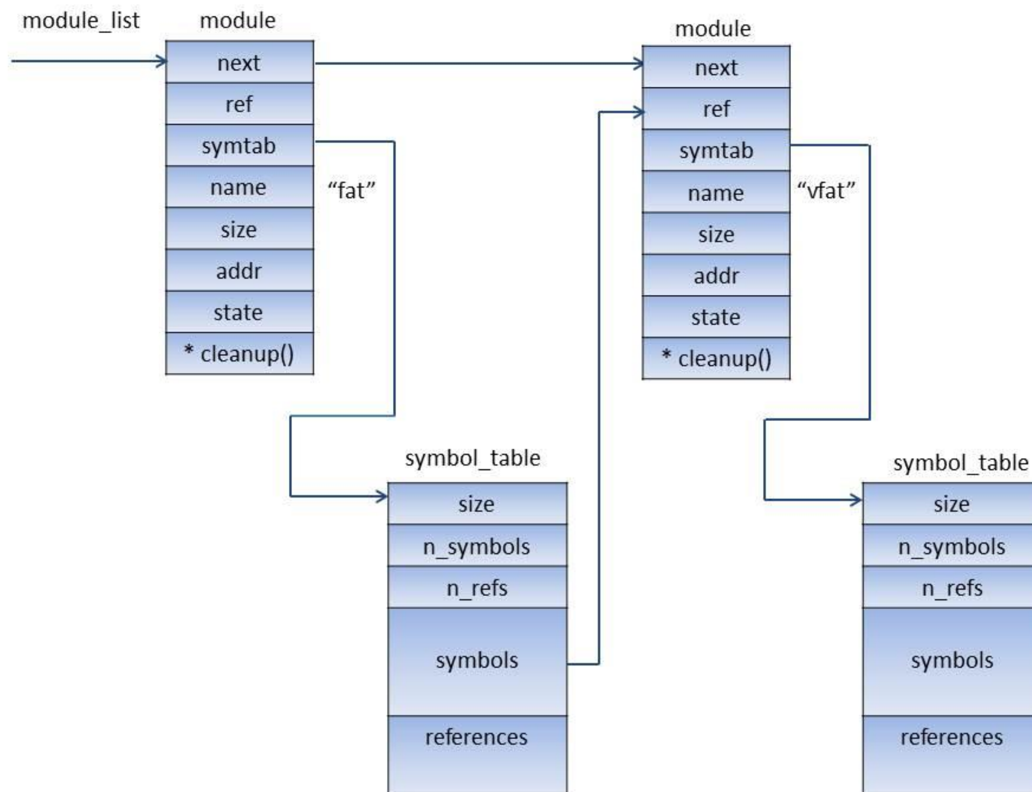

Figura 2.8 Lista de módulos del kernel

En Linux, es posible cargar los módulos de dos formas, la carga manual o la carga automática. Se pueden cargar y descargar explícitamente, de forma manual, usando el comando *insmod* o bien el mismo núcleo puede solicitar que el demonio del núcleo, *kerneld*, cargue y descargue los módulos según los vaya necesitando, esto se conoce como carga bajo demanda.

Cuando el núcleo descubre que necesita un módulo, por ejemplo cuando el usuario monta un sistema de archivos que no está incluido en el núcleo, éste requerirá que el demonio, *kerneld*, intente cargar el módulo apropiado. El demonio del núcleo es un proceso normal de usuario, pero con privilegios de superusuario. Cuando se inicia, normalmente al arrancar el ordenador, abre un canal de comunicación entre procesos, IPC, al núcleo. Este vínculo lo usa el núcleo para enviar mensajes al *kerneld*, solicitando que se ejecuten varias tareas. La labor principal de *kerneld* es cargar y descargar los módulos del núcleo, pero también es capaz de realizar otras tareas como iniciar un enlace *PPP* sobre una línea serie cuando sea necesario y cerrarlo cuando deje de serlo. *Kerneld* no realiza estas tareas por sí mismo, sino que ejecuta los programas necesarios, como *insmod*, para realizar el trabajo. *Kerneld* es sólo un agente del núcleo, planificando el trabajo según su comportamiento.

La utilidad *insmod* debe encontrar el módulo del núcleo requerido que se va a cargar. Los módulos del núcleo cuya carga ha sido solicitada bajo demanda se guardan normalmente en */lib/modules/kernel-version*. Los módulos del núcleo son ficheros objeto vinculados igual que

otros programas en el sistema, con la excepción de que son vinculados como imágenes reubicables. Es decir, imágenes que no están vinculadas para ser ejecutadas desde una dirección específica.

Insmod realiza una llamada al sistema con privilegios para encontrar los símbolos exportados pertenecientes al núcleo. Éstos se guardan en pares conteniendo el nombre del símbolo y su valor, por ejemplo, su dirección. La tabla del núcleo de símbolos exportados se mantiene en la primera estructura de datos *module* en la lista de módulos mantenida por el núcleo, y manteniendo un puntero desde *module_list*. Sólo los símbolos introducidos específicamente se añaden a la tabla, la cual se construye cuando el núcleo se compila y enlaza, en vez de que cada símbolo del núcleo se exporte a sus módulos.

Pueden verse fácilmente los símbolos exportados del núcleo y sus valores echando un vistazo a */proc/ksyms* o usando la utilidad *ksyms*. *Ksyms* puede mostrar todos los símbolos exportados del núcleo o sólo aquellos símbolos exportados por los módulos cargados. *Insmod* lee el módulo en el interior de su memoria virtual y fija las referencias propias hacia las rutinas del núcleo y recursos que estén sin resolver usando los símbolos exportados desde el núcleo. Escribe físicamente la dirección del símbolo en el lugar apropiado en el módulo. Cuando ha arreglado las referencias al módulo convirtiéndolas en símbolos del núcleo exportados, pide al núcleo que le asigne espacio suficiente para el nuevo núcleo, usando, de nuevo, una llamada del sistema privilegiada. El núcleo ubica una nueva estructura de datos *module* y suficiente memoria del núcleo para conservar el nuevo módulo y lo pone al final de la lista de módulos del núcleo. El nuevo módulo se queda marcado como *UNINITIALIZED*.

La figura 2.8 muestra la lista de módulos del núcleo después de que dos módulos, *fat* y *vfat* han sido cargados en el núcleo. No se muestra el primer módulo de la lista, que es un pseudo-módulo que sólo está allí para conservar la tabla de símbolos exportados del núcleo.

Se puede usar el comando *lsmod* para listar todos los módulos del núcleo cargados, así como sus interdependencias. *Lsmod* simplemente reformatea */proc/modules*, que se construye partiendo de la lista de estructuras de datos del núcleo *module*. La memoria que el núcleo utiliza para esto se puede visualizar en el espacio de dirección de memoria del proceso *insmod*, de forma que pueda acceder a ella.

Insmod copia el módulo en el espacio utilizado y lo reubica, de forma que se ejecutará desde la dirección del núcleo en la que ha sido colocado. Esto debe ser así, ya que el módulo no puede esperar ser cargado dos veces en la misma dirección, y aún menos si se trata de dos sistemas Linux distintos. De nuevo, este proceso de reubicación hace necesario que se ajuste la imagen del módulo con la dirección apropiada. El nuevo módulo también exporta símbolos al núcleo, e *insmod* construye una tabla de estas imágenes exportadas.

Cada módulo del núcleo debe contener rutinas de inicialización y limpieza, estos símbolos no son exportados deliberadamente, pero *insmod* debe conocer sus direcciones para así poder pasárselas al núcleo. Si todo va bien, *insmod* ya está listo para inicializar el módulo y hace una

llamada privilegiada al sistema pasando al núcleo las direcciones de las rutinas de inicialización y limpieza del módulo.

Al añadir un módulo nuevo al núcleo, éste debe actualizar el conjunto de símbolos y modificar los módulos que están siendo usados por el módulo nuevo. Los módulos que tienen otros módulos que dependen de ellos, deben mantener una lista de referencias al final de su tabla de símbolos, con un puntero a ella lanzado desde su propia estructura de datos *module*. La figura 2.8 muestra que el módulo de soporte *vfat* depende del módulo de soporte *fat*. Así, el módulo *fat* contiene una referencia al módulo *vfat*, la referencia fue añadida cuando se cargó el módulo *vfat*.

El núcleo llama a la rutina de inicialización de módulos y, si todo funciona correctamente, continúa con la instalación del módulo. La dirección de la rutina de limpieza del módulo se almacena en su propia estructura de datos *module*, a la que el núcleo llamará cuando ese módulo esté descargado. Finalmente, el estado del módulo se establece en *RUNNING*.

Por lo tanto la carga automática inserta el código dinámicamente, según se necesite, esto es atractivo ya que impide que el tamaño del núcleo crezca, además de hacerlo muy flexible. De este modo podemos tener un núcleo reducido que cargue automáticamente los módulos necesarios por el kernel y que cuando detecte que ya no son necesarios los descargue del sistema.

Linux permite el apilamiento de módulos, que es cuando un módulo requiere los servicios de otro módulo. Por ejemplo, el módulo *vfat* requiere los servicios del módulo *fat*, ya que *v fat* es más o menos un conjunto de extensiones *fat*.

Un módulo requiriendo servicios o recursos de otro módulo es muy similar a la situación donde un módulo requiere servicios y recursos del mismo núcleo. Sólo que aquí los requeridos están en otro módulo, que ya ha sido previamente cargado. Mientras se carga cada módulo, el núcleo modifica la tabla de símbolos del núcleo, añadiendo a ésta todos los recursos o símbolos exportados por el módulo recién cargado. Esto quiere decir que, cuando el siguiente módulo se carga, tiene acceso a los servicios de los módulos que ya están cargados.

2.3.3 Descarga de módulos

Los módulos pueden quitarse usando el comando *rmmod*, pero *kerneld* elimina automáticamente del sistema los módulos cargados mediante demanda cuando ya no se usan. Cada vez que su tiempo de inactividad se acaba, *kerneld* realiza una llamada al sistema solicitando que todos los módulos cargados mediante demanda que no estén en uso se eliminen del sistema. El valor del tiempo de inactividad se establece cuando se inicia *kerneld*.

Un módulo no puede ser descargado mientras otros componentes del núcleo dependan de él. Observando la salida de *lsmod*, veremos que cada módulo tiene un contador asociado. Por ejemplo:

Module:	*#pages:*		*Used by:*
msdos	*5*		*1*
vfat	*4*		*1 (autoclean)*
fat	*6*	*[vfat msdos]*	*2 (autoclean)*

El contador indica el número de entidades que dependen de este módulo. En el ejemplo anterior, los módulos *vfat* y *msdos* dependen ambos del módulo *fat*, de ahí que el contador sea 2. Los módulos *vfat* y *msdos* tienen 1 dependencia, que es relativa a un sistema de archivos montado. Si yo tuviera que cargar otro sistema de archivos *vfat*, entonces el contador del módulo *vfat* pasaría a ser 2. El contador de un módulo se conserva en el primer "*longword*" de su imagen.

Este campo se sobrecarga significativamente, ya que también conserva los indicadores *AUTOCLEAN* y *VISITED*. Estos dos indicadores se usan para módulos cargados bajo demanda. Estos módulos se marcan como *AUTOCLEAN* para que el sistema pueda reconocer cuáles puede descargar automáticamente. El indicador *VISITED* marca el módulo como que está en uso por uno o más componentes del sistema, esta marca se establece en cualquier momento que otro componente hace uso del módulo.

Cada vez que *kerneld* pregunta al sistema si puede eliminar módulos cargados bajo demanda, éste inspecciona todos los módulos del sistema buscando candidatos idóneos. Sólo se fija en los módulos marcados como *AUTOCLEAN* y en el estado *RUNNING*. Si el candidato no tiene marcado el indicador *VISITED*, entonces eliminará el módulo, de lo contrario quitará la marca del indicador *VISITED* y continuará buscando el siguiente módulo que haya en el sistema.

Teniendo en cuenta que un módulo puede ser descargado, las llamadas a su rutina de limpieza se producen para permitir liberar los recursos del núcleo que éste ha utilizado. La estructura de datos *module* queda marcada como *DELETED* y queda desvinculada de la lista de módulos del núcleo. Todos los demás módulos de los que dependa el módulo, tienen sus listas de referencia modificadas de forma que ya no lo tienen como dependiente. Toda la memoria del núcleo que necesita el módulo queda liberada.

Cuando se intenta descargar un módulo, el núcleo necesita saber que el módulo no está en uso y necesita alguna manera de notificarle que va a ser descargado. De esa forma, el módulo podrá liberar cualquier recurso del sistema que ha usado, como por ejemplo memoria del núcleo o interrupciones, antes de ser quitado del núcleo. Cuando el módulo está descargado, el núcleo quita cualquier símbolo que hubiese sido exportado al interior de la tabla de símbolos del núcleo.

2.4 Drivers

Un *driver* o controlador de dispositivos realiza la interfaz de comunicación con el hardware, dice al kernel como se usa un determinado hardware. Es el programa que sirve para poder manejar de forma cómoda unos dispositivos, es decir permite, que el hardware y la forma de utilizarlo sean transparentes al usuario. Es algo bastante oscuro y específico, que trata directamente con el hardware. Cuando instalamos un driver, el kernel queda modificado permanentemente, pues ahora contiene además de lo que contenía antes, la información específica de nuestro nuevo hardware.

Los controladores de dispositivos, *device drivers,* son implementados como módulos del núcleo. Son programas añadidos al núcleo del sistema operativo, concebidos inicialmente para gestionar periféricos y dispositivos especiales. Los controladores de dispositivo, permiten añadir nuevos componentes al ordenador sin necesidad de recompilar el sistema operativo. Son semejantes a la idea de subsistema en un sistema operativo con micronúcleo, reciben y realizan las operaciones.

Haciendo una descomposición por capas tendríamos un esquema como el siguiente:

Figura 2.9 Descomposición por capas

Por tanto, un **driver** es un software formado por un conjunto de rutinas y tablas que, instalado, forman parte del sistema operativo, y sirve para ejecutar y controlar todas las operaciones de E/S que se realizan sobre el periférico conectado a la computadora y que controla dicho driver. Es un conjunto de programas que provee la interface entre el sistema operativo y un tipo de dispositivo periférico.

En términos generales, el trabajo de un driver es aceptar pedidos abstractos del software independientes del dispositivo que está por encima de él y controlar que se ejecuten.

En Linux todos los dispositivos instalados están mapeados en memoria existiendo para cada uno de ellos un archivo en el directorio ***/dev*** que representa a dicho dispositivo. Pueden ser accedidos a través de una interfaz de ficheros bajo ese directorio. Lógicamente, un driver debe dar ciertas funcionalidades para poder manipular ese archivo, y en consecuencia el dispositivo, mediante algunas funciones.

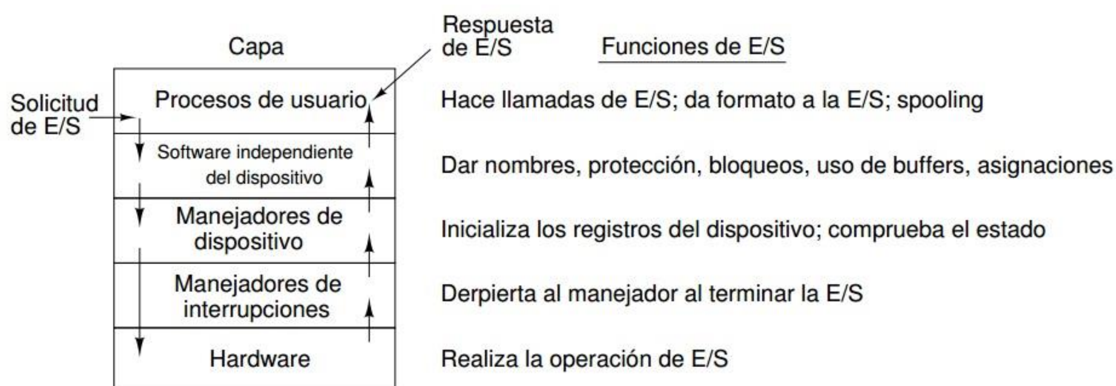

Figura 2.10 Esquema Global de controladores de dispositivos

2.4.1 Tipos de drivers

Existen varios tipos de controladores de dispositivos, los ficheros asociados a un dispositivo pueden ser dispositivos de tipo **carácter** o de tipo **bloque**. Existen otros tipos específicos, como los interfaces de red, drivers para dispositivos USB, SCSI, módulos serie, tarjetas Wireless, etc.

La diferencia fundamental entre los dispositivos orientados a caracteres y los orientados a bloques es que los primeros reciben o envían la información carácter a carácter; en cambio, los controladores de dispositivo de bloques procesan, como su propio nombre indica, bloques de cierta longitud en bytes, sectores. Se diferencian ambos tipos de dispositivos por el modo de acceso que tiene el kernel a ellos y en la interfaz entre el kernel y el propio dispositivo.

La mayoría de los dispositivos del mundo son de carácter, porque no necesitan este tipo de *buffering*, y no operan con un tamaño de bloque fijo.

Sabremos de que tipo es un dispositivo mirando el primer carácter de la salida de "*ls –la*". Si es "*b*" entonces es un dispositivo de bloque, y si es "*c*" es un dispositivo de carácter.

2.4.1.1 Dispositivos de bloque

Estos dispositivos se caracterizan por ser de acceso aleatorio y por bloques, la unidad mínima de lectura / escritura es un bloque, aceptan bloques de entrada y de salida cuyo tamaño puede variar según el dispositivo. Almacenan la información en bloques de tamaño fijo, los datos se transmiten en forma de bloques, cada uno con su propia dirección. Los tamaños comunes de los bloques van desde 128 bytes hasta 1024 bytes, en Linux suelen ser de 512 ó 1024 bytes. Utilizan el buffer caché del sistema, acceden directamente a los inodos del sistema de ficheros en el directorio */dev* y tiene relación directa con el sistema de ficheros.

La propiedad esencial es que en todo momento el programa puede leer o escribir en cualquiera de los bloques. En estos dispositivos, el sistema operativo mantiene un vector de dispositivos de bloque llamado vector *blkdevs*. Cada entrada de este vector está formada por la dirección de la rutina de atención al dispositivo y una lista de peticiones. Cada entrada de la lista de peticiones, representa el buffer asociado a esa petición para realizar la transacción de datos.

Tienen un buffer para las peticiones, por lo que pueden escoger en qué orden las van a responder, es importante en caso de dispositivos de almacenamiento (es más rápido leer o escribir sectores que están cerca entre sí).

Usualmente suelen ser dispositivos rápidos como pueden ser unidades de almacenamiento, por ejemplo los discos duros, los disquetes, los CDROMS, etc.

2.4.1.2 Dispositivos de carácter

Se encarga de la transferencia de caracteres entre el dispositivo físico y la aplicación de usuario. Son dispositivos de acceso secuencial, pueden direccionarse a nivel de byte. Estos dispositivos envían o reciben un flujo de caracteres o de bytes, sin sujetarse a una estructura de bloques. La información se transmite carácter a carácter, no se pueden utilizar direcciones ni tienen una operación de búsqueda. No utilizan el buffer caché del sistema.

El sistema operativo crea el vector *chrdevs* en el que cada entrada es una estructura formada por un puntero al número de versión mayor y otro a la lista de operaciones para ese dispositivo. En cada una de las operaciones aparece otro puntero con la dirección de memoria en la que está la citada función. El vector de dispositivos de carácter, vector *chrdevs*, está ordenado por los números de versión mayor. Al inicializar el driver, éste crea una entrada en el vector *chrdevs* para el dispositivo. Pueden ser accedidos como si de un archivo se tratase, hay definidas un tipo de operaciones sobre ellos muy similares a las de ficheros, *file_operations*, en las que estarán definidas operaciones tan comunes como read, write, open, close, etc.

Algunos ejemplos de dispositivos de carácter son el ratón, el teclado, un terminal de texto, la consola (*/dev/console*), una cinta magnética, los puertos serie (*ttys* o *pttys*), interfaces de red, una impresora (*/dev/lp0*) y cualquier dispositivo cuyo acceso sea a través de flujos de bytes.

2.4.1.3 Dispositivos de red

Permiten el control de los datos transferidos a través de interfaces de red. La comunicación con el kernel es bastante distinta a los casos anteriores. Manejan los diferentes protocolos de red existentes.

2.4.1.4 Referenciar los dispositivos

Los dispositivos tienen que tener una forma de ser referenciados, la manera de tenerlos referenciados es a través de dos números que se corresponden con el **número mayor** y el **menor**. El número mayor corresponde al dispositivo al que hace referencia y el menor es la forma de catalogarlo dentro de ese tipo como dispositivo, un ejemplo sería que el número mayor 1 corresponda con el ratón y dentro del ratón habrá distintos tipos que los distinguiremos a través del número menor.

Cuando accedemos a un dispositivo, hemos de utilizar su driver asociado, el número mayor, no es más que un identificador que nos informa de por qué driver es manejado cada dispositivo. Como un mismo driver puede manejar varios dispositivos, es necesario identificar estos con otro número, el número menor. Así, un determinado dispositivo está determinado unívocamente por su combinación "número mayor-número menor".

Gracias a la existencia de los números menores, un mismo driver puede implementar diferentes operaciones dependiendo del número menor del dispositivo que lo esté usando. Por ejemplo el driver de los dispositivos *null* y *zero* es el mismo, pero dependiendo de por cuál de los dos le llames, se comportará de manera diferente.

Con los kernels 2.4 se introduce una nueva forma de hacer referencia a los dispositivos, a través de *devfs*, que lo que hace es utilizar un sistema de ficheros para almacenar los nombres de los dispositivos y de esa forma se hará referencia, así se consigue no tener que tener cargado en memoria toda la lista de */dev* por parte del sistema de ficheros sino que devfs se encargará de hacer referencia a cada dispositivo por cada nombre y además se consigue no tener que asignar un espacio de memoria por cada dispositivo sino que únicamente la parte relativa a lo que ocupa *devfs*.

Para crear un fichero especial de dispositivo se usa la orden *mknod*, que recibe un nombre de fichero, un tipo (bloques o caracteres), un *major* y un *minor* number. Para hacer corresponder un *mayor* y *minor number* al dispositivo lo que hay que hacer es registrar el nuevo dispositivo que vayamos a crear, eso se hace con:

- Para dispositivos de carácter:

```
int register_chrdev (unsingned int mayor, const char
*name, struct file_operations *fops);
```

- Para dispositivos de bloque:

```
int register_blkdev (unsingned int mayor, const char
*name, struct file_operations *fops);
```

Al hacer esto habremos creado nuestro dispositivo, nos bastará con consultar el */proc/devices*. Los parámetros que se le pasan se corresponden al *inodo* mayor que es el tipo de dispositivo, podemos mirar la tabla en la documentación en las fuentes del kernel, el nombre del dispositivo y *fops* se corresponde a un puntero a las operaciones que podremos definir sobre nuestro dispositivo, podemos ver su estructura en el objeto *file_operations* relativos al sistema de ficheros virtual, *VFS*, en */usr/include/Linux/vfs.h*. Aunque la estructura *fops* es la misma existen diferencias a la hora de usarla según el tipo de dispositivo.

3 Llamadas a procedimientos remotos (RPC)

Este capítulo tiene como objetivo describir las llamadas a procedimientos remotos, las cuales permiten la comunicación de procesos en un sistema distribuido. Se exponen los mecanismos de comunicación existentes tanto a bajo como a alto nivel, para centrarnos en el utilizado en la implementación del proyecto, las RPC.

A través de la historia y por medio de una introducción a las RPC´s nos encaminamos a la terminología utilizada para ahondar en los *stubs*, en el formato de presentación y en el lenguaje de definición de interfaces.

Se plantea cómo es el modelo utilizado en las RPC´s para su comprensión, se atiende al protocolo de transporte utilizado, al modelo de comunicación entre un cliente y un servidor con RPC y su semántica de fallos. Finalmente se profundiza en las RPC de SUN y cómo programar con ellas, abordando los requisitos del protocolo, qué las caracteriza, cómo dotar de autenticación a los mensajes y asignar el número de programa así como la necesidad de usar el portmapper.

Por último se expone el lenguaje RPC de SUN, necesario para poder realizar desarrollos con el protocolo de red, se termina mostrando un ejemplo de comunicación entre cliente y servidor usando rpcgen.

3.1 Mecanismos de comunicación en sistemas distribuidos

Uno de los principales objetivos de un sistema distribuido es poder comunicar procesos entre sí. Existen diferentes mecanismos de comunicación en un sistema distribuido:

- **Mecanismos de bajo nivel**: el programador tiene que preocuparse de establecer la comunicación, la representación de los datos, etc. Los métodos de comunicación de este tipo más utilizados son:

    1. Colas de mensaje: se basa en la comunicación de procesos a través de mensajes, los cuales llegan a “buzones”, colas de mensaje. Estos mensajes se recogen o se dejan en la cola compartida, conocida por el cliente y el servidor. El envió es asíncrono pero la recepción puede ser sincronía o asíncrona.
    2. Sockets: un *socket* es un punto final de comunicación, que ofrece una interfaz de acceso a los servicios del nivel de transporte de los protocolos TCP/IP. Se identifica por una dirección IP y un número de puerto con el que se puede utilizar el protocolo TCP y UDP. EL protocolo TCP es un protocolo orientado a conexión, como primer paso de la comunicación se realiza un envío de paquetes entre los procesos que se van a comunicar, para asegurarla y además es fiable, intenta recuperarse en caso de fallo de la conexión. UDP en cambio es un protocolo no orientado a conexión y no fiable, ni realiza una primera conexión, directamente envía los datos, ni intenta recuperarse en caso de fallos en la conexión.

- **Mecanismos de alto nivel**: ofrecen abstracciones donde el programador no debe preocuparse de establecer los protocolos. Ejemplos de este tipo de comunicación:

1. Llamadas a procedimientos remotos: es un híbrido entre llamadas a procedimientos y paso de mensajes. El cliente invoca los procedimientos que se encuentran implementados en el servidor, de forma similar a como se haría con llamadas a procedimientos locales. La comunicación es más transparente que en los mecanismos de bajo nivel. Un ejemplo son las RPC de SUN.
2. Invocación de métodos remotos (entornos orientados a objetos): similar al anterior con la diferencia de que se encuentran en un lenguaje orientado a objetos. Por ejemplo, RMI en Java y CORBA.

3.2 Remote procedure call (RPC)

A continuación se describe brevemente las llamadas a procedimientos remotos, RPC o *Remote Procedure Call*, tanto su historia como su definición. Se incluye un ejemplo para comprender su funcionamiento.

3.2.1 Historia

Las llamadas a procedimientos remotos proporcionan una mayor comodidad que el paso de mensajes debido a que proporciona una interfaz al más alto nivel. El modelo de RPC fue propuesto por Birrel y Nelson en 1985. La idea era enmascarar un sistema distribuido usando una abstracción transparente debido a que:

- Parece una llamada a procedimiento local.
- Esconde todos los aspectos relacionados con la comunicación de procesos en sistemas distribuidos.
- Permite un modelo de programación de aplicaciones distribuidas más sencillo.

Las RPC son la base para muchas aplicaciones y sistemas distribuidos. Posteriormente aparecen las SUN RPC, que son la base para varios servicios actuales como NFS o NIS. Llegaron a su culminación en 1990 con DCE *(Distributed Computing Enviroment)* de OSF. En la actualidad han evolucionado hacia orientación a objetos, con la invocación de métodos remotos, por ejemplo, Corba y RMI.

3.2.2 Introducción

RPC es un mecanismo para el desarrollo de aplicaciones cliente-servidor, basada en la invocación de procedimientos remotos. Las ***llamadas a procedimientos remotos*** son funciones que se invocan desde el cliente y se ejecutan en el servidor. A continuación, el servidor devuelve la respuesta al cliente, de manera que la comunicación se realiza de la forma más transparente posible. El objetivo es que un programador pueda desarrollar las funciones entre los clientes y el servidor.

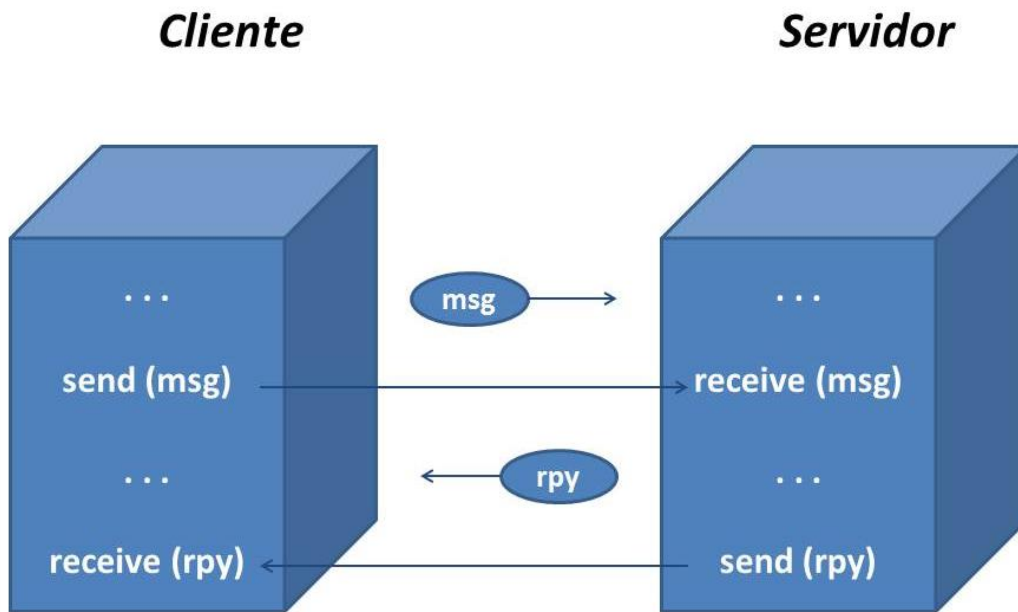

Figura 3.1 Paso de mensajes (visión de bajo nivel)

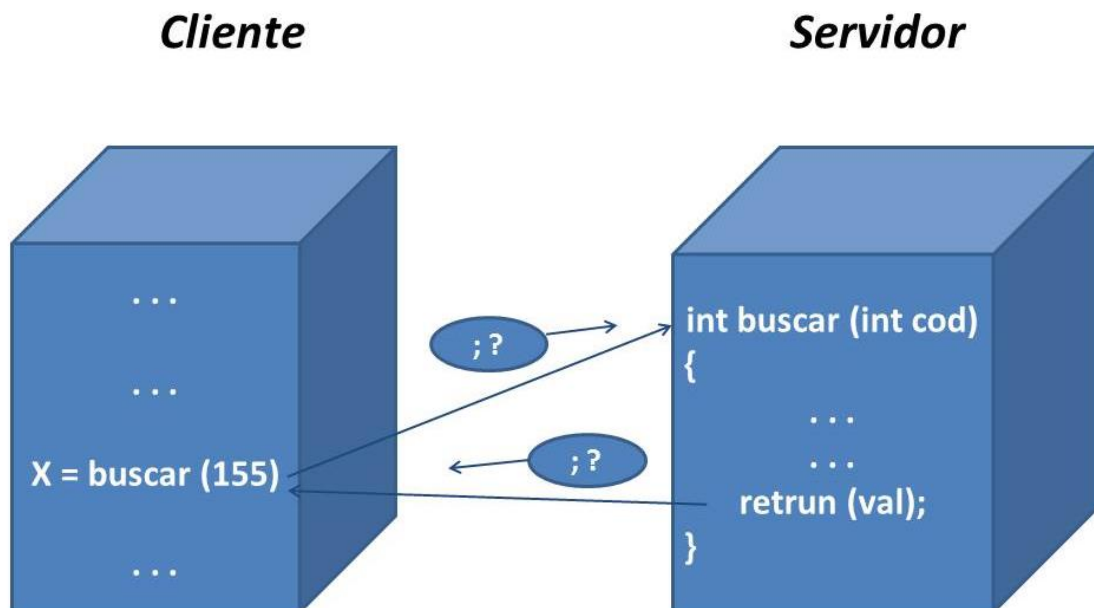

Figura 3.2 Llamadas a procedimientos remotos

El funcionamiento general de una RPC puede ser explicado desde dos puntos de vista, el del cliente y el del servidor. En el cliente se ejecuta el proceso que realiza la llamada a la función. Dicha llamada empaqueta los argumentos en un mensaje y se los envía a otro proceso, quedando a la espera del resultado. Las figuras 3.1 y 3.2 muestran este proceso.

En el extremo contrario, el servidor recibe un mensaje consistente en varios argumentos. Estos argumentos son usados para llamar a otra función del servidor. El resultado de la función se empaqueta en un mensaje que se retransmite al cliente.

El objetivo final, como ya he comentado, es acercar la semántica de las llamadas a procedimiento convencional a un entorno distribuido, lo que conlleva el concepto de transparencia.

Los elementos necesarios son los siguientes:

- Código del cliente
- Código del servidor
- Formato de representación
- Definición de la interfaz
- Localización del servidor
- Semánticas de fallo

En cuanto a la ejecución, lo ideal sería que no hubiera distinción entre la ejecución remota o la local de una función. Sin embargo, existen dos problemas esenciales que impiden que esta transparencia sea completa:

- La latencia de red: la invocación de una función remota tardara varios órdenes de magnitud más que la invocación local.
- La fiabilidad de la red: la invocación remota implica enviar al servidor la identificación de la funciona a ejecutar, al igual que los parámetros, y recibir del servidor el resultado de la ejecución. Cualquiera de estos mensajes, solicitud o respuesta, pueden ser extraviado o duplicado por la red, por lo que deberá considerarse el manejo de estos errores en las aplicaciones.

3.2.3 Terminología

Un ***servidor*** es un proceso que lleva a cabo servicios de red. Un ***servicio de red*** es una colección de uno o más programas remotos. Un ***programa*** lleva a cabo uno o más *procedimientos remotos*. Un ***procedimiento remoto*** es un procedimiento cuya petición lanza el cliente y su ejecución la realiza el servidor.

Los ***clientes*** son procesos que llaman a *servicios* a través de *procedimientos remotos*. Un *servidor* puede aportar más de una versión de un *programa* para ser compatible con las definiciones de los *procedimientos remotos* anteriores.

Los ***stubs*** o suplentes son los que se encargan de la comunicación entre los procesos. El *suplente* del cliente se encarga de empaquetar los parámetros de la petición, enviarlos al *suplente* del servidor, y luego esperar la respuesta, desempaquetarla y entregarlas al cliente. El *suplente* del servidor se encarga de esperar las peticiones, desempaquetar los datos enviados desde el cliente, invocar el procedimiento local, recoger los resultados de éste, empaquetar estos resultados en un nuevo mensaje y enviarlos al cliente.

Un aspecto importante del aplanamiento o ***marshalling*** es la representación de los datos, puesto que el cliente y el servidor pueden ejecutar en computadoras de arquitecturas distintas donde los datos pueden representarse de forma diferente. En este caso es necesario utilizar alguna convención para que el cliente y el servidor puedan entender sin problemas los datos. Una posibilidad es enviar los datos por la red en un formato estándar y unificado, de forma que cuando se envían los datos, estos se convierten al formato estándar y cuando se reciben se convierten del formato estándar al formato computador.

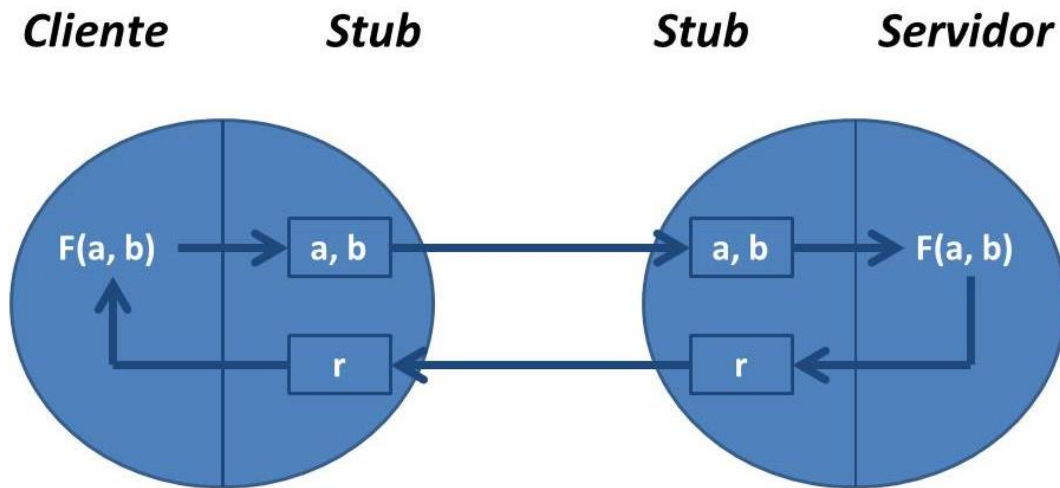

Figura 3.3 Proceso de empaquetamiento y envió

3.2.3.1 Resguardos (Stubs o suplentes)

Las funciones de abstracción de una llamada RPC a intercambio de mensajes se denominan suplentes o stubs. Se generan automáticamente por el software RPC en base a la interfaz del servicio. Son independientes de la implementación que se haga del cliente y del servidor. Solo dependen de la interfaz.

Entre las tareas que realizan están las de localizar al servidor, empaquetar y construir mensajes, enviar el mensaje al servidor y esperar la recepción del mensaje y devolver los resultados. Se basan en una librería de funciones RPC para las tareas más habituales. En la siguiente figura se observan los procesos que intervienen en una llamada RPC.

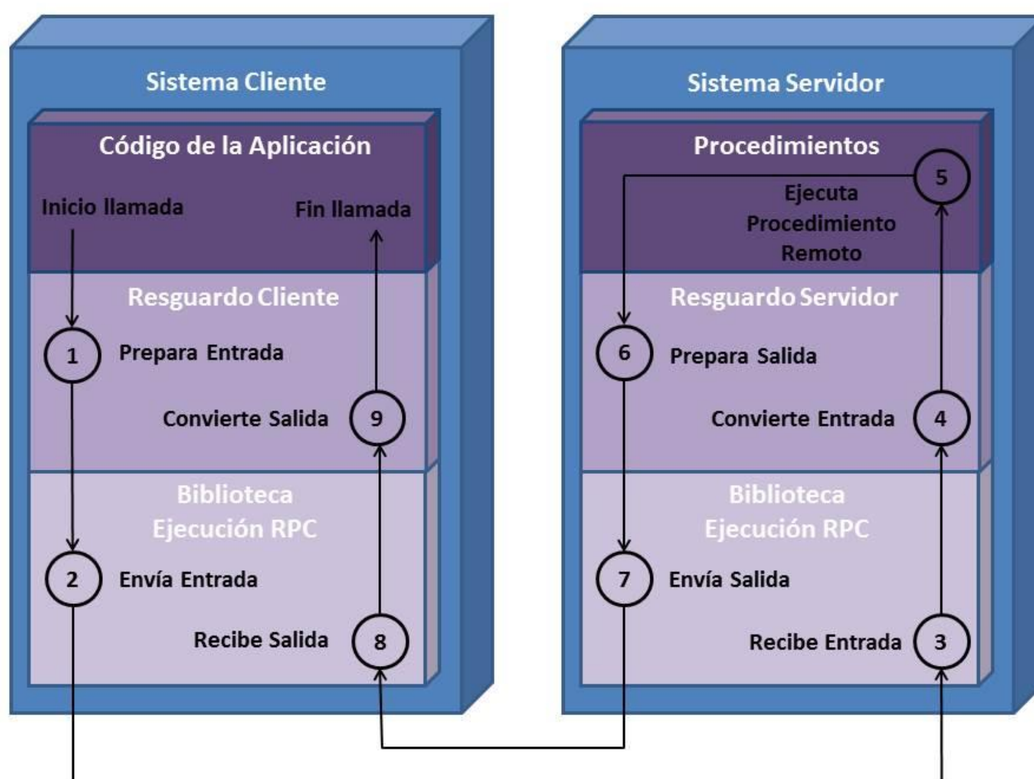

Figura 3.4 Código del cliente y del servidor

3.2.3.2 Formato de representación

Una de las funciones de los suplentes es empaquetar los parámetros en un mensaje, el llamado *aplanamiento* o *marshalling*. Se tienen que evitar los problemas en la representación de los datos. Por ejemplo, servidor y cliente pueden ejecutar en máquinas con arquitecturas distintas. XDR, ***eXternal Data Representation***, es un estándar que define la representación de tipos de datos. Hay que tener cuidado con el paso de parámetros de entrada / salida. Por ejemplo, existen problemas con los punteros, ya que una dirección solo tiene sentido en un espacio de direcciones determinado.

3.2.3.3 Lenguaje de definición de interfaces

IDL, ***Interface Definition Language***, es un lenguaje de representación de interfaces. Hay muchas variantes de *IDL* en función del tipo de RPC. Puede estar integrado con un lenguaje de programación (Cedar o Argus) o ser específico para definir las interfaces (RPC de SUN y RPC de DCE). El *IDL* define procedimientos y argumentos, no así la interpretación. Se usa habitualmente para generar de forma automática los suplentes.

3.2.4 El modelo de RPC

El modelo utilizado en las *llamadas a procedimientos remotos* es similar al modelo usado en las *llamadas a procedimientos locales*. En el caso de llamadas locales, se pasan los argumentos a un procedimiento, se pasa el control al procedimiento, se recogen los resultados del procedimiento y se continúa la ejecución. La llamada a procedimiento remoto es similar, solo

que en vez de un proceso con el control ahora existen dos procesos, uno es el proceso que realiza la llamada al procedimiento y el otro es el proceso servidor que implementa el procedimiento. Es decir, el proceso que realiza la llamada envía un mensaje al proceso servidor y espera, se bloquea, hasta recibir un mensaje de respuesta. El mensaje de petición contiene los parámetros del procedimiento, entre otras cosas. El mensaje de respuesta contiene los resultados del procedimiento, entre otras cosas. Una vez recibido el mensaje de respuesta, se extraen los resultados del procedimiento, y se reanuda la ejecución del proceso cliente.

El funcionamiento del modelo, véase figura 3.5, es el siguiente:

- El servidor registra su servicio en el sistema y se queda esperando la llegada de una petición.
- El cliente, debe localizar o "enlazarse" al servicio que desea utilizar a través de la mediación de un *binder* o enlazador dinámico.
- El cliente empaqueta los datos, argumentos del procedimiento, en una petición y la envía al servidor.
- El proceso servidor recoge los parámetros del procedimiento, ejecuta la llamada y devuelve los resultados empaquetados, enviando un mensaje de respuesta, para después quedarse esperando la siguiente petición.

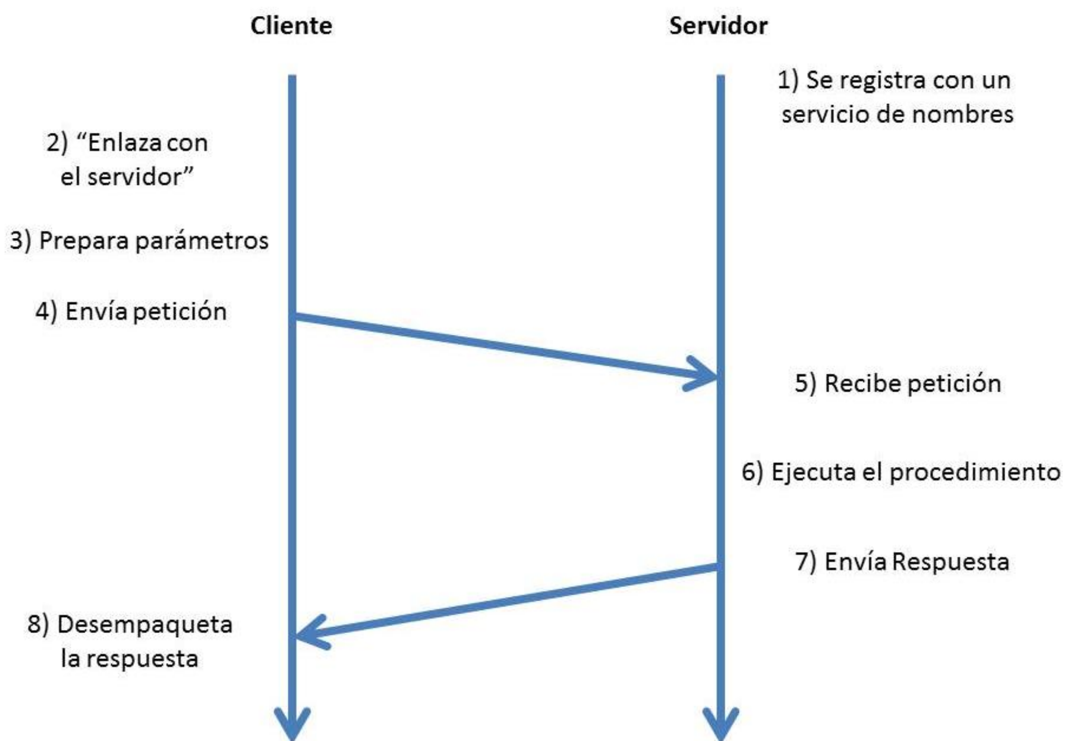

Figura 3.5 Esquema de una petición RPC

3.2.5 Protocolo de transporte

El protocolo RPC es independiente del protocolo de transporte utilizado. Es decir, RPC no se preocupa de la forma en que un mensaje se pasa de un proceso a otro. El protocolo realiza la especificación e interpretación de mensajes. Es importante señalar que las RPC no intentan llevar a

cabo cualquier tipo de fiabilidad y que la aplicación debe ser consciente del tipo de protocolo de transporte que existe en el nivel inferior. Por ejemplo, si está funcionando sobre un transporte fiable como TCP, entonces la mayoría del trabajo ya lo realiza este protocolo de transporte, ya que el protocolo TCP se encarga de que el mensaje llegue al destino.

Por otro lado, si está utilizando un transporte no fiable como UDP, se debe llevar a cabo una política de reenvío y de "*tiempo de espera*" ya que este protocolo de transporte no lo controla. Debido a la independencia de transporte, el protocolo de RPC no une la semántica específica a los procedimientos remotos o su ejecución.

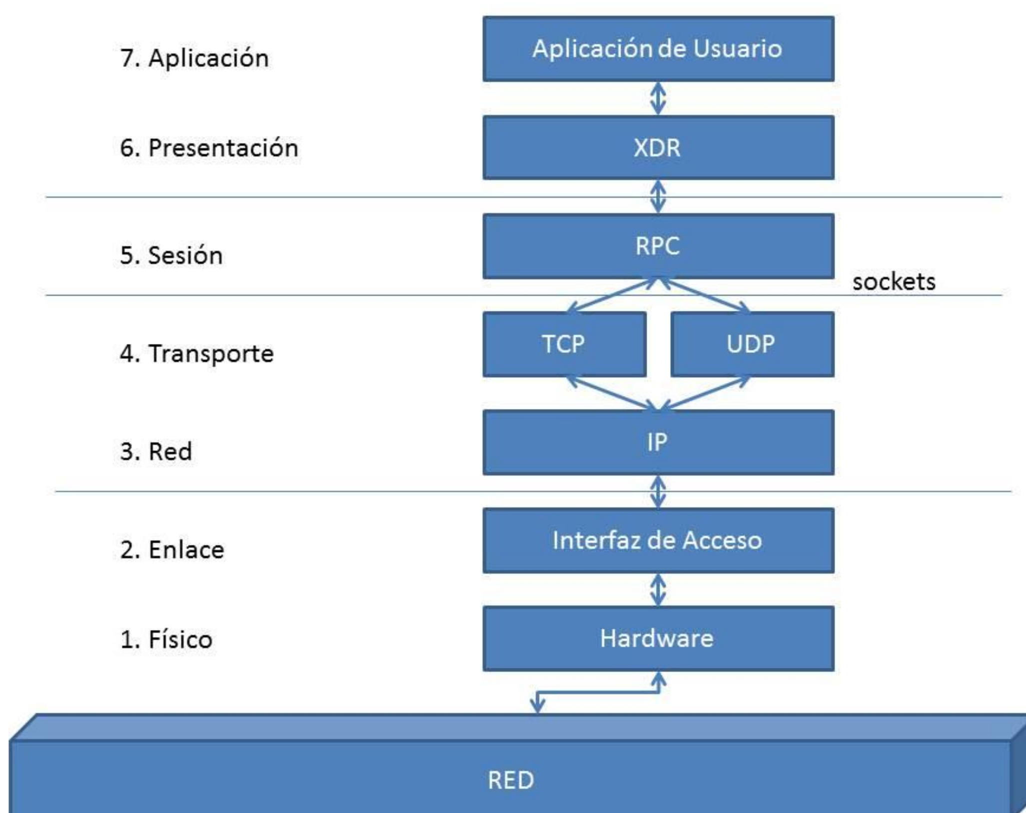

Figura 3.6 Arquitectura del modelo RPC

3.2.6 Localización del servidor

La comunicación de bajo nivel entre cliente y servidor por medio de paso de mensajes, por ejemplo con sockets, requiere dos tareas: una la de **localizar** la dirección del servidor, tanto dirección IP como número de puerto en el caso de sockets, y otra la de **enlazar** con dicho servidor. Estas tareas las realiza el resguardo del cliente.

El acto de enlazar un cliente a un servicio no es parte de la especificación de *llamadas a procedimientos remotos*. Esta función importante y necesaria queda en manos de algún software del alto nivel.

En el caso de servicios cuya localización no es estándar se recurre al enlace dinámico. Este permite localizar objetos con nombre en un sistema distribuido, en concreto, servidores que

ejecutan las RPC. El "*enlazador dinámico de nombres*" o ***binder*** es el servicio que permite la comunicación entre el cliente y el servidor. Hay dos tipos de enlace:

- Enlace no persistente: la conexión entre el cliente y el servidor se establece en cada llamada RPC.
- Enlace persistente: la conexión se mantiene después de la primera RPC. Es útil en aplicaciones con muchas RPC repetidas, aunque como contrapartida da problemas si los servidores cambian de lugar o fallan.

Para realizar el enlace dinámico se requiere de un *binder* o **enlazador dinámico**. El *binder* es el servicio que mantiene una tabla de traducciones entre nombres de servicio y direcciones. Incluye funciones para registrar un nombre de servicio (y su versión si es necesario), eliminar un nombre de servicio, y buscar la dirección correspondiente a un nombre de servicio.

El enlazador dinámico es un elemento conocido por los componentes del sistema distribuido que ejecuta en una dirección fija de un computador fijo. El sistema operativo se encarga de indicar su dirección difundiendo un mensaje *broadcast* cuando los procesos comienzan su ejecución. La anterior figura 3.6 muestra el protocolo básico RPC.

3.2.7 Modelo genérico de comunicación

El modelo de comunicación entre un cliente y un servidor con RPC, es el siguiente:

- El servidor registra los servicios que tiene implementados en el enlazador.
- El cliente realiza una petición de búsqueda del servicio, al cual se desea acceder, al enlazador (servidor de nombres).
- Este devuelve la dirección al cliente, donde se encuentra el servidor.
- El cliente y el servidor ya se pueden comunicar entre sí.

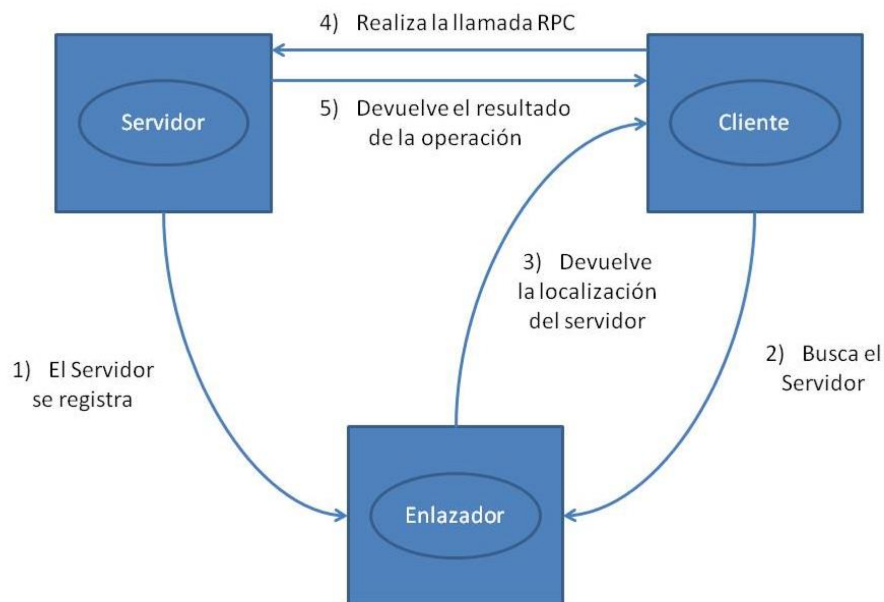

Figura 3.7 Modelo de comunicación

3.2.8 Semántica de Fallos

Los problemas que puede plantear un cliente son los siguientes:

- El cliente no puede localizar un servidor:

  Varias pueden ser las razones para que esto ocurra. La primera y más obvia es la de que el servidor esté caído. Otro problema frecuente es que el cliente está usando una versión antigua del servidor. Para evitar esto se incluye la versión en el binder. La versión ayuda a detectar accesos a copias obsoletas. El error se puede indicar al cliente de dos maneras diferentes. La devolución de un código de error es una de ellas, aunque tiene como contrapartida que no es transparente. La segunda opción es elevar una excepción, para lo que se necesita un lenguaje que tenga excepciones.

- Se pierde el mensaje de petición del cliente al servidor:

  Es la más fácil de tratar. Es suficiente con la activación de un timeout, tiempo de espera, después del mensaje. Si no se recibe una respuesta se retransmite. Depende del protocolo de comunicación subyacente.

- Se pierde el mensaje de respuesta del servidor al cliente:

  Este caso es más difícil de tratar. Se pueden emplear alarmas y retransmisiones, pero no se puede especificar si los que se perdió fue la petición, si fue la respuesta o si únicamente el servidor va lento. Algunas operaciones pueden repetirse sin problemas. Son las llamadas operaciones idempotentes. La solución con operaciones no idempotentes es descartar peticiones ya ejecutadas. Se pueden identificar con un número de secuencia en el cliente o con un campo en el mensaje que indique si una petición es original o es una retransmisión.

- El servidor falla después de recibir una petición:

  En este caso puede que el servidor haya llegado a ejecutar la operación o no lo haya hecho. En este último caso se podría retransmitir, pero el cliente no puede distinguir entre las dos. En este caso se debe decidir lo que hacer. Las opciones son:

  - No garantizar nada.
  - Semántica de al menos una vez:
    Reintentar y garantizar que la RPC se realiza al menos una vez. No vale para operaciones no idempotentes.
  - Semántica de a lo más una vez:
    No reintentar, puede que no se realice ni una sola vez
  - Semántica de exactamente una:
    Sería lo deseable

- El cliente falla después de enviar una petición:

  En este caso la computación está activa pero ningún cliente espera los resultados. Supone gasto de ciclos de CPU y además si el cliente rearranca y ejecuta de nuevo la RPC se pueden crear confusiones.

La semántica puede inferirse del protocolo de transporte subyacente. Por ejemplo, considérese que la RPC trabaja encima de un transporte no fiable como UDP. Si una aplicación RPC retransmite los mensajes con tiempos de espera cortos, la única cosa que puede inferirse si no recibe ninguna respuesta es que el procedimiento se ejecutó cero o más veces. Si recibe una respuesta, entonces puede inferir que el procedimiento fue ejecutado "por lo menos una vez". Un servidor puede desear recordar peticiones que previamente realizó un cliente y no recordar el orden de estas peticiones para asegurar en algún grado la semántica de ejecutar "al menos una vez". Un servidor puede hacer esto aprovechándose del identificador de la petición que se empaqueta con cada petición RPC.

El identificador de petición se crea en el cliente, por lo que, este puede escoger enviar otra petición con otro identificador distinto. El proceso servidor sabe este hecho y puede escoger recordar este identificador después de realizar una petición y no admitir peticiones con el mismo identificador para lograr algún grado de semántica "ejecutar al menos una vez".

Al servidor no se le permite examinar el identificador excepto para verificar que el identificador no se ha utilizado anteriormente. Por otro lado, si se usa un transporte fiable como TCP, la aplicación puede inferir con la recepción por parte del cliente de un mensaje de respuesta, que el procedimiento fue ejecutado precisamente una vez, pero si no recibe ningún mensaje de respuesta, no puede asumir que el procedimiento remoto no fue ejecutado.

Aun cuando un protocolo orientado a conexión como TCP se usa, una aplicación todavía necesita tiempos de espera y reenvíos para manejar caídas del servidor. Hay otras posibilidades para transportes además del uso de datagramas o protocolos orientado a conexión.

3.3 RPC de SUN

A continuación se describe uno de los estándares de RPC más usados, las RPC de SUN utilizadas para la comunicación con servidores NFS, *Network File System* o sistema de ficheros en red.

Las RPC de SUN usan como lenguaje de definición de interfaz (IDL) una interfaz que contiene un número de programa y un número de versión. Cada procedimiento especifica un nombre y un número de procedimiento. Como característica fundamental de los procedimientos está el que reciben un único parámetro. De igual manera, los parámetros de salida se devuelven mediante un único resultado. El lenguaje ofrece una notación para definir constantes, tipos, estructuras, uniones y programas.

Usa como compilador *rpcgen*. Con él se compilan las interfaces y se generan el suplente del cliente, el suplente del servidor y el procedimiento principal del servidor, procedimientos para

el aplanamiento/des aplanamiento (marshalling), y un fichero de cabecera (.h) con los tipos y declaración de prototipos.

En lo referente al enlace dinámico, el cliente debe especificar el *host* en el que ejecuta el servidor. El servidor se registra (n° de programa, n° de versión y n° de puerto) en el *portmapper* local. El proceso se completa cuando el cliente envía una petición al *portmapper* del *host* donde ejecuta el servidor.

3.3.1 Programación con RPC

El programador debe proporcionar la definición de la interfaz, es decir, el fichero IDL. En él se debe facilitar el nombre de las funciones, los parámetros que el cliente pasa al servidor y los resultados que el servidor debe devolver al cliente. Además, el programador tiene que definir el código del cliente y el código del servidor.

El compilador, como ya se ha comentado antes, proporciona el suplente del cliente y el suplente del servidor. Todo el proceso se resume en la figura.

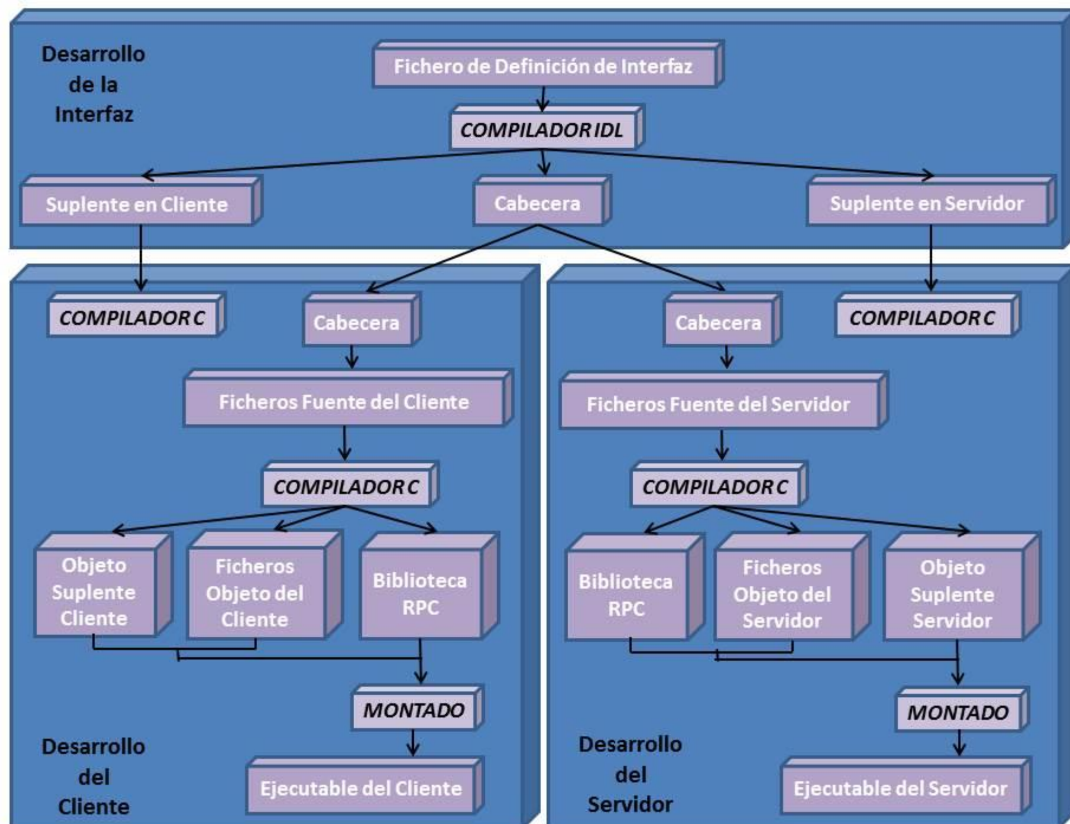

Figura 3.8 Programación con RPC

3.3.2 Requisitos del protocolo RPC de SUN

El protocolo de RPC tiene los siguientes requisitos:

1. Solo se puede llamar a una única especificación de un procedimiento.
2. Previsión de envío de mensajes de respuesta ante mensajes de petición.
3. Previsión de autenticación de la petición y respuesta RPC.

3.3.3 Programas y procedimientos

Las RPC se caracterizan por 3 campos:

- El número del programa.
- Número de versión del programa.
- El número del procedimiento remoto.

Los tres campos identifican el procedimiento a ejecutar. Los números de programa son administrados por una autoridad central como Sun o IANA. Una vez que la interfaz tiene un número programa, se puede llevar a cabo la implementación de su programa; la primera aplicación tendría el número de versión igual a 1 probablemente. El campo de versión permite al cliente utilizar diversas versiones de un *procedimiento remoto* de un mismo servidor, sin tener que cambiar el cliente para que funcione con la nueva versión. De manera que pueden existir diversas versiones definidas en la interfaz de un mismo programa. El número del procedimiento identifica el procedimiento a ser invocado. En conjunto para un programa pueden existir varias versiones de un mismo procedimiento remoto.

Ejemplo:

Estructura de la petición:

```
struct mis_param {
      int  a;
      int  b;
};
```

Esta es la interfaz de un programa, en este caso tiene dos versiones para poder ver la existencia de un mismo procedimiento remoto en cada una de estas.

```
program calcular
{
 version UNO
      {
          int sumar(mis_param param) = 1;
         } = 1;
 version DOS
      {
          int sumar(mis_param param) = 1;
           int restar(mis_param param) = 2;
         } = 2;
} = 100022254;
```

El mensaje de respuesta a una petición puede dar los siguientes errores:

1. La aplicación remota tiene otra versión del protocolo.
2. El programa no está disponible en el sistema remoto.
3. El programa no se corresponde con el número de versión pedido.
4. El número del procedimiento pedido no existe.
5. Los parámetros al procedimiento remoto parecen ser basura desde punto de vista del servidor.

3.3.4 Autenticación

La autenticación de los mensajes la proporciona el mismo protocolo RPC. El mensaje de la llamada tiene dos campos de autenticación, las credenciales, que identifican al cliente, y verificador, un campo que sirve para verificar la identidad del cliente. El mensaje de la respuesta tiene un campo de autenticación, el verificador de respuesta, verificador que indica que el que ha respondido es realmente el servidor al cual se realizó la petición. La especificación del método de autenticación se realiza mediante las siguientes estructuras:

```
enum authflavor{
      AUTHNULL = 0,
      AUTHUNIX = 1,
      AUTHSHORT = 2,
      AUTHDES = 3
};

struct opaqueauth{
      authflavor  flavor;
      opaque  body<400>;
};
```

La estructura *opaqueauth* es un conjunto de datos en los cuales se indica tanto el método de autenticación como los datos propios de la *llamada al procedimiento remoto.* La interpretación y semántica de los datos contenidos dentro de los campos de la autenticación son especificadas individualmente por cada programa, la autenticación es independiente las especificaciones del protocolo.

Si se rechazan los parámetros de autenticación, la respuesta contiene información en la cual se informa por qué fueron rechazados estos parámetros.

3.3.5 La asignación del número de programa

Los números del programa se reparten en grupos de 0x20000000 (decimal 536870912) según el mapa siguiente:

Dirección	**Descripción**
0 – 1fffffff	Definidos por SUN
20000000 - 3fffffff	Definidos por usuario
40000000 - 5fffffff	Variable
60000000 - 7fffffff	Reservado
80000000 - 9fffffff	Reservado
a0000000 - bfffffff	Reservado
c0000000 - dfffffff	Reservado
e0000000 - ffffffff	Reservado

El primer grupo es un rango de números administrado por Sun Microsystems y debe ser idéntico para todos los sistemas. El segundo rango es para las aplicaciones. Este rango está pensado principalmente

para preparar nuevos programas. El tercer grupo es para las aplicaciones que generan el número de programa dinámicamente. Los grupos finales son reservados para uso futuro, y no deben ser utilizados.

3.3.6 Otros usos del protocolo RPC

El uso intencional de este protocolo es invocar procedimientos remotos. Es decir, cada mensaje de petición se empareja con un mensaje de respuesta. Sin embargo, el propio protocolo es un protocolo de paso de mensajes, que puede ser implementado con otros protocolos distintos a las RPC.

Sun usa, el protocolo RPC para los dos siguientes protocolos (no RPC):

- *batching* (o *pipelining)*
- *broadcast* RPC.

3.3.6.1 Batching

Batching le permite a un cliente enviar una sucesión arbitrariamente grande de mensajes a un servidor; el batching usa protocolos de transmisión fiables (como TCP) para su transporte. En el caso de batching, el cliente nunca espera una respuesta del servidor, y el servidor no envía respuestas a las peticiones. Una sucesión de llamadas normalmente es terminada por un RPC de aceptación.

3.3.6.2 Broadcast RPC

En el broadcast el cliente envía un paquete de broadcast a la red y espera las numerosas respuestas de cada uno de los servidores. Broadcast RPC usa protocolos de transporte no orientados a conexión, con paquetes basados en protocolos tipo UDP. Los servidores que mantienen el protocolo broadcast responden cuando la petición se procesa con éxito, y no responden en caso de error.

3.3.7 Protocolo del programa PortMapper

El enlazador dinámico o el *portmapper* hacen las correspondencias entre los programas RPC y los números de versión con puertos específicos. Este programa enlaza dinámicamente los programas remotos si le es posible. Esto es deseable ya que el rango de números del puerto reservados es muy pequeño y el número de potenciales programas remotos es muy grande. El *portmapper* se ejecuta en un puerto reservado, los números de puerto de otros programas remotos puede ser determinados preguntando al *portmapper.*

El *portmapper* también ayuda en la comunicación RPC. Un programa RPC dado, normalmente tendrá un número de puerto distinto en diferentes máquinas, así que no hay ninguna manera de transmitir directamente a todos estos programas. El *portmapper,* sin embargo, tiene un número del puerto fijo. Así que, para transmitir a un programa dado, el cliente envía su mensaje realmente al *portmapper.* Cada *portmapper* que recoge la transmisión entonces llama al servicio local especificado por el cliente. Cuando el *portmapper* recibe la respuesta del servicio local, envía la respuesta devuelta al cliente.

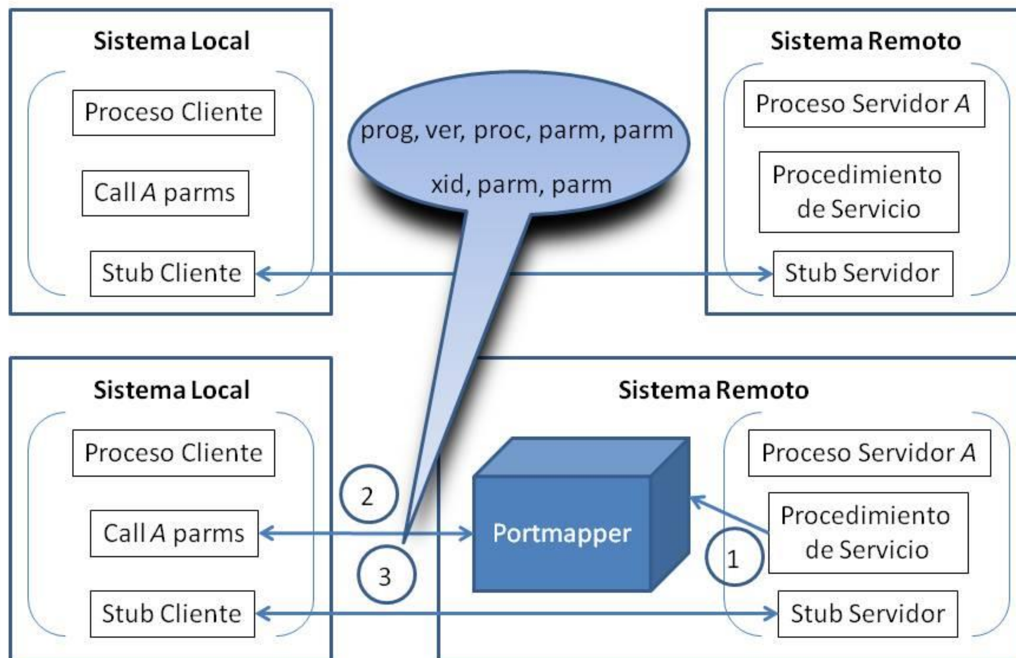

Figura 3.9 Especificación del protocolo con el lenguaje RPC

3.3.7.1 Especificación del protocolo con el lenguaje RPC

```
const PMAPPORT = 111; //Número del puerto del portmapper
/*El contenido  (program, version, protocol) del
número del puerto */
struct mapping {
      unsigned int prog;
      unsigned int vers;
      unsigned int prot;
      unsigned int port;
};

// Valores soportados por el portmapper
const IPPROTOTCP = 6 ;  // Número de protocolo para TCP/IP
const IPPROTOUDP = 17; // protocol number for UDP/IP

// Lista de contenidos
struct *pmaplist {
      mapping map;
      pmaplist next;
};

// Argumentos de la llamada
struct callargs {
      unsigned int prog;
      unsigned int vers ;
```

```
        unsigned int proc ;
        opaque argso;
};

// Resultados de la llamada
struct callresult {
        unsigned int port;
        opaque res<>;
};

// Procedimientos del portmapper
program PMAPPROG {
        version PMAPVERS {
                void PMAPPROCNULL(void) = 0;
                bool PMAPPROCSET( mapping) = 1;
                bool PMAPPROCUNSET (mapping) = 2;
                unsigned int PMAPPROCGETPORT (mapping) = 3;
                pmaplist PMAPPROCDUMP(void) = 4;
                callresult PMAPPROCCALLIT (callargs) = 5;
        } = 2;
} = 100000;
```

3.3.7.2 Operaciones del PortMapper

El programa *portmapper* actualmente soporta dos protocolos, UDP y TCP. El *portmapper* utiliza el puerto 111, SUNRPC, en cualquiera de estos protocolos. Lo siguiente es una descripción de cada uno de los procedimientos del *portmapper:*

- PMAPPROCNULL: este procedimiento no hace nada. Por convención, el procedimiento cero de cualquier programa no toma ningún parámetro, ni devuelve ningún resultado.
- PMAPROCSET: Cuando un programa se quiere utilizar en una máquina, lo primero que tiene que hacer es registrar en el *portmapper* de la misma máquina. El programa pasa su número del programa, número de la versión, protocolo de transporte, y el puerto en el que espera peticiones de servicio. El procedimiento devuelve una respuesta de tipo booleano cuyo valor es TRUE si el procedimiento estableció la conexión con éxito y FALSE en cualquier otro caso. El procedimiento no realiza la operación si ya existe otro elemento con los mismos valores de programa, versión, y protocolo.
- PMAPPROCUNSET: cuando un programa se hace inaccesible, debe borrar su registro en el *portmapper* de la misma máquina. Los parámetros y resultados tienen significados idénticos a los de PMAPPROCSET. Se ignoran los campos del protocolo y del número de puerto del argumento.
- PMAPPROCGETPORT: dado un número del programa, número de la versión, y el número del protocolo de transporte, este procedimiento devuelve el número del puerto en el que el programa está esperando peticiones de servicio. Un valor de puerto O indica que el programa no ha sido registrado. El campo del puerto se ignora de los argumentos.
- PMAPPROCDUMP: este procedimiento enumera todas las entradas en la base de datos del *portmapper.* El procedimiento toma ningún parámetro y devuelve una lista de valores y cada elemento contiene un programa, una versión, un protocolo, y un puerto.

- PMAPPROCCALLIT: este procedimiento permite llamar a un procedimiento remoto en la misma máquina sin saber el número del puerto del procedimiento remoto. Se piensa para las transmisiones de apoyo a los programas remotos arbitrarios vía el *portmapper* con puerto conocido. Los parámetros *prog, vers, proc,* y los bytes de *args* son el número del programa, número de la versión, número del procedimiento, y parámetros del procedimiento remoto.

Nota:

- Este procedimiento sólo envía una respuesta si el procedimiento fue ejecutado con éxito y está en "silencio", no contesta, en cualquier otro caso.
- El portmapper se comunica con el programa mediante UDP.

El procedimiento devuelve el número del puerto del programa, y los bytes de los resultantes son los resultados del procedimiento remoto.

3.4 Lenguaje RPC de SUN

En el protocolo NFS, los datos enviados a través de mensajes RPC, deben ser representados bajo un formato entendible tanto por el emisor como el receptor. XDR permite representar de forma común, los tipos de datos de toda la vida, tales como, enteros y string, así como estructuras de datos algo más complejas, arrays, listas enlazadas y uniones, entre otras, para permitir la comunicación entre máquinas que tienen representaciones de datos distintas.

El estándar XDR, asume que 8 bits-bytes, u octetos, son portables y define cada tipo de dato como una secuencia de bytes. XDR utiliza una representación de los datos canónica, tal que cada tipo de dato tiene una única representación en código XDR Cualquier programa que necesite transmitir datos a otra máquina (a través de una red, o mediante otros medios como disquetes ó cintas magnéticas) puede convertir su representación de datos local en codificación XDR. Igualmente, el programa que recibe estos datos, a partir de los datos que recibe codificados en XDR, obtiene datos entendibles por la máquina local.

El lenguaje RPC es una ampliación del lenguaje XDR. La ampliación consiste en la introducción de prototipo del programa. A continuación se van a comentar el lenguaje RPC y XDR.

3.4.1 Definiciones

El lenguaje RPC consiste en una seria de definiciones.

```
lista-de-definiciones:
     definición ";"
     definición ";" lista-de-definiciones
```

Se reconocen 5 tipos de definiciones.

```
definición:
      enum-definition // Definición de enumerados
      struct-definition // Definición de estructuras
      union-definition // Definición de uniones
      typedef-definition // Definición de tipos
      const-definition // Definición de constantes
      program-definition // Definición de programa
```

Los ejemplos que aparecen para cada una de las definiciones de tipos están acompañados por el código generado con el *rpcgen*, programa generador de *stubs*.

3.4.2 Estructuras

La definición una estructura en el lenguaje XDR es igual que una en C.

```
"struct" identificador "{"
      lista-de-definiciones;
"}"

lista-de-definiciones:
      definición ";"
      definición ";" lista-de-definiciones
```

Ejemplo:

Código XDR	Código C

`struct coord`
`{`
`int x;`
`int y;`
`};`
	`struct coord`
	`{`
	`int x;`
	`int y;`
	`};`
	`typedef struct coord coord;`

3.4.3 Uniones

Las uniones XDR son discriminantes:

```
union-definition:
      "unión"  identificador  "switch"  "("declaración")"
      "{"
            lista-case
      "}"

lista-case:
      "case" valor ":" declaracion ";"
      "default" ":" declaracion";"
      "case" valor ":" declaracion ";" lista-case
```

Ejemplo:

Código XDR	Código C

union readresult switch (int errno) {
case 0: opaque data [1024];
default: void;
};
	struct readresult {
	int errno;
	unión {
	char data[1024];
	} readresult_u;
	};
	typedef struct readresult readresult;

3.4.4 Enumerados

Los enumerados del lenguaje XDR se escriben igual que los enumerados en C.

```
enum-definition:
      "enum" idenficador "{"
            lista-valores-enum
      };

lista-valores-enum:
      valor-enum
      valor-enum "," lista-valor-enum

valor-enum:
      identificador-valor-enum
      identificador-valor-enum "=" value
```

Ejemplo:

Código XDR	Código C

enum colortype {
RED = 0,
GREEN = 1,
BLUE = 2
}
	enum colortype {
	RED = 0,
	GREEN = 1,
	BLUE = 2,
	}
	typedef enum colortype colortype;

3.4.5 Tipos

Es la misma sintaxis que para los tipos en C.

```
definicion-tipos:
      "typedef" definición
```

Ejemplo:

Código XDR	Código C
typedef string fnametype <255>;	typedef char *fnametype;

3.4.6 Constantes

Las constantes en XDR son iguales a las constantes definidas en C. Pero las constantes solo pueden ser enteros.

```
definición-constante:
      "const" identificador "=" integer
```

Ejemplo:

Código XDR	Código C
Const DOCE = 12;	#define DOCE 12

3.4.7 Programas

Los programas tienen la siguiente sintaxis:

```
definicion-programa:
      "program" identificador "{"
            lista-version
      "}" "=" valor

lista-version:
      versión ";"
      versión ";" lista-version

versión:
      "versión" identificador-version "{"
            lista-procedimiento
      "}" "=" valor

lista-procedimiento:
      procedimiento ";"
      procedimiento ";" lista-procedimiento

procedimiento:
      identificador-tipo identificador-procedimiento "(
            identificador-tipo
      ")" " = " valor
```

Ejemplo:

```
program TIMEPROG {
      version TIMEVERS {
            unsigned int TIMEGET (void) = 1 ;
            void TIMESET(int) = 2 ;
      } = 1;
} = 44;
```

Los nombres de los procedimientos, de la versión, y del programa pasan a ser constantes en el fichero con extensión ".h" generado:

```
#define TIMEPROG 44
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

Las cabeceras de los procedimientos generadas al compilar la interfaz, son diferentes a la cabecera del procedimiento de la interfaz, tanto en el cliente como en el servidor.

Por ejemplo el procedimiento “sumar” que se encuentra en la interfaz pasa a ser dos procedimientos:

Interfaz:

```
program suma
{
 version UNO
      {
          int sumar(mis_param param) = 1;
         } = 1;
} = 100022254;
```

Función del cliente:

```
extern  int * sumar_1(mis_param , CLIENT *);
```

Como se puede ver, la llamada de la función en el cliente se compone de un valor de retorno que es un puntero a un entero. Esto es debido a que devuelve NULL en caso de perder la comunicación. El nombre del procedimiento viene seguido de su número de versión. Luego los parámetros a la función son:

- El puntero a la estructura de datos que se envía al servidor.
- El puntero a la conexión con el servidor.

Función del servidor:

```
extern  int * sumar_1_svc(mis_param , struct svc_req *);
```

Es similar a la interfaz generada para el cliente, con la diferencia de que *svc* indica que se utiliza en el servidor y la estructura *struct svc_req* que es una estructura para que el *stub* del servidor se comunique con la implementación del procedimiento remoto del servidor.

3.4.8 Declaraciones

En lenguaje XDR los únicos tipos de declaraciones son:

```
declaración:
      declaración-simple
      declaracion-array-fijo
      declaracion-array-variable
      declaration-puntero
```

1. **Declaración simple:** se declara como en C.

```
declaración-simple:
     identificador-tipo identificador-de-la-variable
```

Ejemplo:

Código XDR	Código C
`colortype color;`	`colortype color;`

2. **Declaración de array de longitud fija** es como la declaración en C:

```
declaracion-array-fijo:
     identificador-tipo     identificador-de-la-variable
     "["valor"]"
```

Ejemplo:

Código XDR	Código C
`colortype palette[8];`	`colortype palette[8];`

3. **Declaración de array de longitud variable** No existe en C como tal, pero existe en XDR.

```
declaracion-array-variable:
     identificador-tipo identificador-de-la-variable
     "<" valor ">"
     identificador-tipo identificador-de-la-variable
     "<" ">"
```

El número indica que el *array* tiene un máximo. Si no tiene número entonces es que no existe longitud.

```
int heights<12>;/* como mucho 12 elementos */
int widths<>; /* cualquier número de elementos */
```

En C no existen explícitamente los *array* de tamaño variable pero se pueden simular.

Ejemplo:

Código XDR	Código C

`int heights<12>;`	`struct {`
	`uint heightslen;`
	`int *heightsval;`
	`} heights;`

El número de elementos que hay en el puntero, está indicado por el entero de la estructura.

4. **Declaración de punteros:** Se declaran igual que en C, pero un puntero en XDR se utiliza para pasar árboles y listas,... Se pueden pasar datos, pero no referencias, ya que en cada máquina se tiene una referencia distinta.

```
Declaración-puntero:
    identificador-tipo "*"  identificador-de-la-variable
```

Ejemplo:

Código XDR	Código C
`listitem *next;`	`listitem *next;`

3.4.9 Casos especiales

Existen unas pocas excepciones a las reglas anteriores:

- **Booleans**: En C no existe el tipo de datos boolean. Sin embargo, la librería RPC tiene un tipo de datos llamado boolt que es igual que el tipo de datos boolean y toma valores TRUE o FALSE.

Ejemplo:

Código XDR	Código C
`bool married;`	`boolt married;`

- **Strings**: C no tiene un tipo de datos string, en cambio utiliza un conjunto de bytes de tipo char terminados con un carácter nulo o vacío como carácter de fin de cadena. En el lenguaje XDR, los strings son declarados usando la palabra reservada string y al compilarlo pasa a ser un puntero a una cadena de caracteres en el fichero que contiene las cabeceras. El tamaño máximo de un string aunque puede ser definido (como aparece en los ejemplos), este valor no se tiene en cuenta ya que no existe ningún delimitador al tamaño.

Ejemplo:

Código XDR	Código C
`string name<32>`	`char *name;`
`string longname<>;`	`char *longname;`

- **OpaqueData**: Un dato opaco es usado en las RPC y XDR para describir un tipo de datos indefinido, que una serie arbitraria de bytes. Puede ser declarado como un array de tamaño fijo o de tamaño variable.

Ejemplo:

Código XDR	Código C
`opaque diskblock[512];`	`char diskblock[512];`

opaque filedata<1024>;	struct { uint filedatalen; char *filedataval; } filedata;

- **Voids**: En una declaración de tipo void la variable no está nombrada. La declaración es void seguido de ";". Las declaraciones de tipo void solo pueden ponerse en:

    - Definiciones de uniones

```
union readresult switch (int errno) {
      case 0: opaque data[1024];
      default: void;
};
```

    - Definiciones de programas (como argumento o resultado de un procedimiento remoto).

```
void TIMESET(int) = 2;
```

3.5 Ejemplo de RPC

A continuación se muestra un ejemplo de comunicación entre un cliente y un servidor, utilizando como método de comunicación las RPC de SUN. El *cliente* realizará una petición de suma de dos números y el servidor devolverá el resultado.

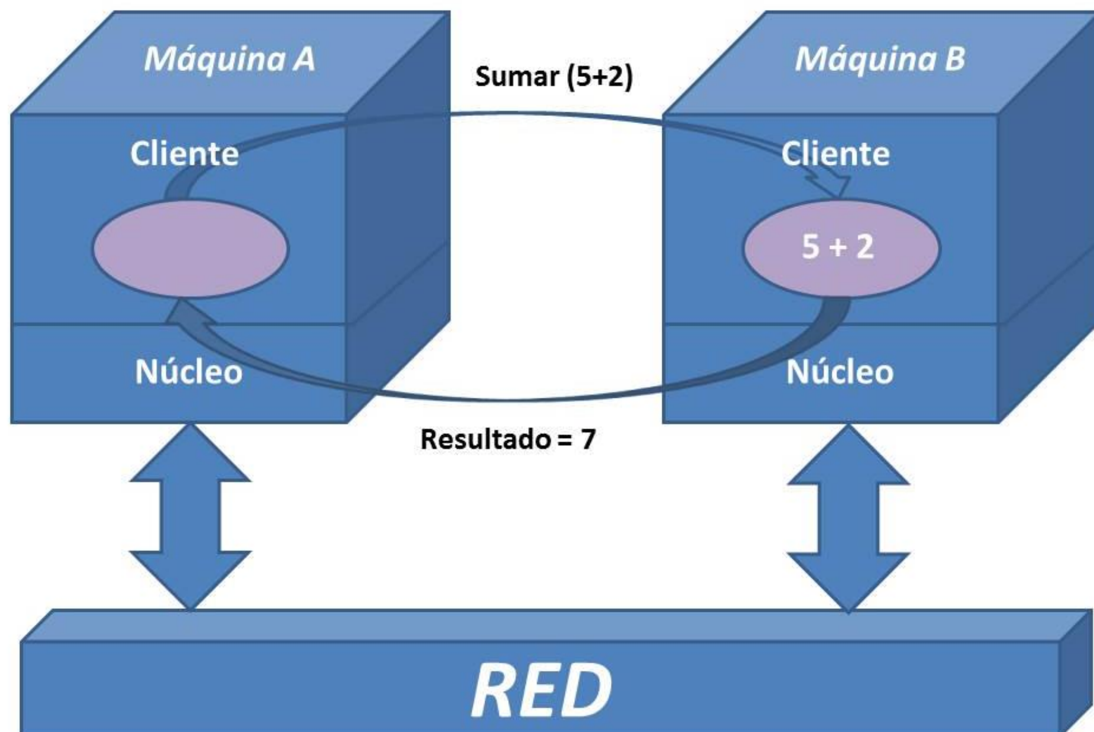

Figura 3.10 Esquema de la comunicación

suma.x incluye la interfaz del servidor. En este fichero se especifica la interfaz del servidor, en la cual se declaran los programas las versiones y los procedimientos remotos:

```
struct mis_param
{
      int a;
      int b;
};

program suma
{
 version UNO
      {
          int sumar(mis_param param) = 1;
         } = 1;
} = 100022254;
```

Al compilar, usando *rpcgen*, la interfaz se obtiene como resultado:

- Un archivo de cabecera con las estructuras y funciones que aparecen en el prototipo.
- Un stub o suplente para el cliente, el cual realizará la conexión con el servidor.
- Un stub o suplente para el servidor, que recibirá las peticiones de los clientes e invocará a los procedimientos.
- Procedimientos para realizar el aplanamiento

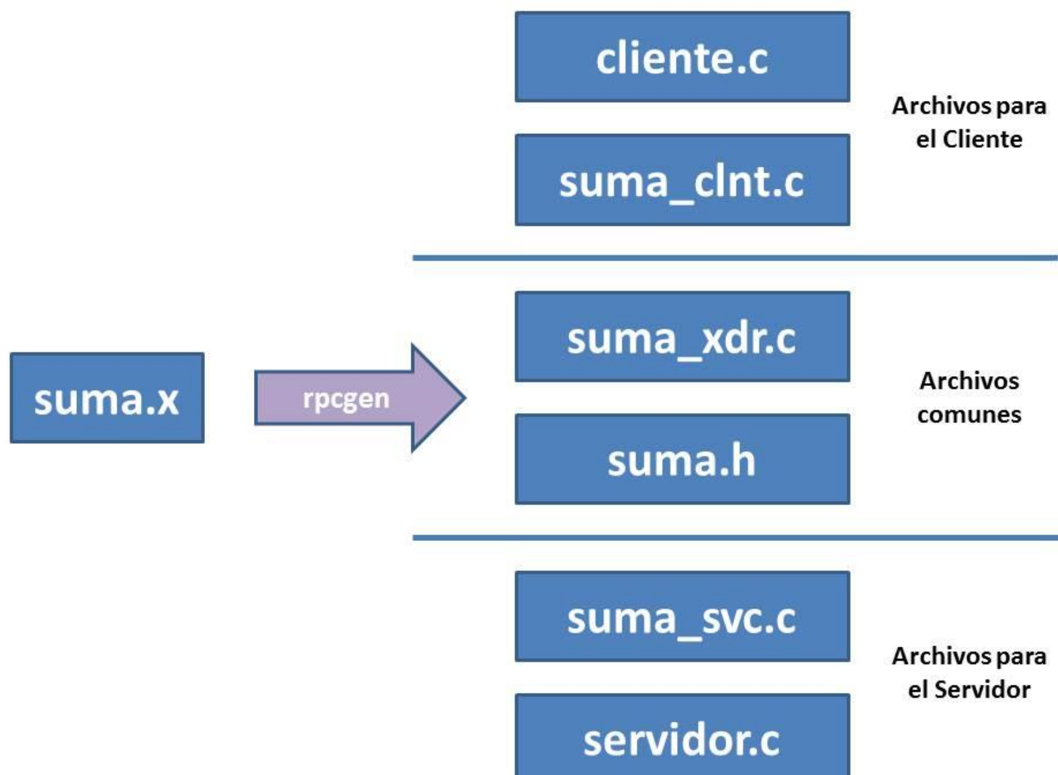

Figura 3.11 Conjunto de ficheros que se utilizan en esta comunicación RPC

Esta es la cabecera que genera en este caso el rpcgen:

suma.h

```
#ifndef _SUMA_H_RPCGEN
#define _SUMA_H_RPCGEN

#include <rpc/rpc.h>

struct mis_param {
      int a;
      int b;
};
typedef struct mis_param mis_param;

#define suma 100022254
#define UNO 1

#define sumar 1
extern  int * sumar_1(mis_param , CLIENT *);
extern  int * sumar_1_svc(mis_param , struct svc_req *);

/* the xdr functions */

extern  bool_t xdr_mis_param (XDR *, mis_param*);

#endif /* !_SUMA_H_RPCGEN */
```

Esta cabecera la comparten tanto el cliente como el servidor ya que son los tipos de datos que se envían.

El código del servidor que realiza la operación de suma es:

servidor.c

```
#include "suma.h"

int *sumar_1_svc(struct mis_param param, struct svc_req
*rqstp)
{
   static int r = 0;
   printf("Realizando la petición del cliente\n");
   r = param.a + param.b;
   printf("r = %d\n",r);
   return(&r);
}
```

Tiene 2 parámetros:

- El primero, contiene los argumentos que recibe para realizar la suma.
- El segundo, contiene una estructura que mantiene la conexión.

Este procedimiento remoto se encuentra en espera hasta recibir una petición.

El código del cliente será el siguiente:

cliente.c

```
#include "suma.h"

void main (int argc, char **argv)
{
  char *host; // variable para almacenar la maquina
  CLIENT *sv; // Almacena el descriptor de la conexión
  int res = 0; // Almacena el resultado
  struct mis_param param; // Almacena los datos a enviar

  if(argc < 2) {
     printf("Error, has de introducir dos parámetros\n");
     exit(-l); // Abandono la ejecución
  }

  host = argv[1];// Introduzco la máquina en la variable

  // Realiza la conexión con la máquina remota
  sv = clnt_create(host, suma, UNO, "tcp");

  if (sv == NULL) // Si hay error
   {
    clnt_pcreateerror(host);
    exit(-1); // Abandono la ejecución
   }

  printf("Ha conectado con el servidor: %s\n",host);
  // Introduzco los valores en la estructura
  param.a = 5;
  param.b = 2;
  res = sumar_1(param, sv);
  if (res == NULL) /*si hay error*/
   {
    clnt_perror(sv, "Error en la llamada");
    clnt_destroy(sv); // Destruyo la conexión
    exit(-1); // Abandono la ejecución
   }
  printf("5 + 2 = %d\n", *res);
  clnt_destroy(sv); // Destruyo la conexión
  exit(0);/*abandonamos la ejecución*/
}
```

Al principio del programa se debe crear una conexión con el servidor al cual se quiere acceder. Esto se realiza mediante la función *clnt_create*. Los parámetros para la creación de la conexión son:

- El nombre o dirección de la maquina en la cual está el servicio.
- El número de programa.
- La versión.
- El protocolo de transporte.

El resultado de la función *clnt_create*, si ha podido crearse la conexión, un puntero a una estructura de tipo *CLIENT*. En caso de error se devuelve *NULL*. La estructura *CLIENT* es una estructura que almacena los datos de la conexión, y debe ser referenciada en cada una de las llamadas RPC que se realicen.

Al realizar una llamada RPC, el cliente recibirá los datos devueltos como un puntero a los datos. Si el valor del puntero de los datos es igual a *NULL*, entonces es que se ha perdido la conexión con el servidor.

Al final de programa se debe realizar una operación de destrucción de la conexión. Esto se hace con la función *clnt_destroy* y tiene como parámetro el puntero a la estructura *CLIENT* que se creó en la llamada *clnt_create.*

3.6 Entornos de objetos distribuidos

La extensión de los mecanismos de RPC a una programación orientada a objetos dio lugar a los modelos de objetos distribuidos. Entre las ventajas está el hecho de que métodos remotos están asociados a objetos remotos, lo cual es un hecho más natural para el desarrollo orientado a objetos. Además admite modelos de programación orientada a eventos. La dificultad mayor a solucionar es el hecho de que la referencia al objeto es algo fundamental. Además los objetos pueden ser volátiles o persistentes.

En este tipo de entornos se usa un *middleware.* Este se define como un nivel de abstracción para la comunicación de los objetos distribuidos. Sus cometidos son ocultar varios elementos como puedan ser la localización de los objetos, los protocolos de comunicación, el hardware usado o los sistemas operativos. También se hace necesario un modelo de objetos distribuidos que describa los aspectos del paradigma de objetos que es aceptado por la tecnología, como puedan ser la herencia, las interfaces, las excepciones, el polimorfismo, etc. Por último, estos entornos suelen tener un sistema de recogida de basuras, que determinan los objetos que no están siendo usados para liberar recursos.

Tecnologías de desarrollo de sistemas distribuidos basados en objetos:

- ANSA (1989- 1991) fue el primer proyecto que intentó desarrollar una tecnología para modelizar sistemas distribuidos complejos con objetos.
- DCOM de Microsoft.
- COREA de OMG.
- Tecnologías de Sun Microsystems:
  - Remote Method Invocation (RMI)
  - Enterprise Java Beans (EJB)
  - Jini
- Diferentes en tornos de trabajo propietarios.

4 Sistemas de ficheros distribuidos

En este capítulo se define qué es un sistema de ficheros distribuido, SFD. Se realiza una pequeña introducción y se muestra una arquitectura tipica del diseño de un sistema de ficheros distribuido, sus principales caracteristicas, componentes y servicios.

Entre los servicios se encuentran los de directorios y los de ficheros, en los que se profunndiza para comprender que es un esquema de nombres, cuales son sus propiedades, cuales son los niveles de traducción y esquemas basicos tanto de los nombres de usuario como de los del sistema para los servicios de directorios. Se expecifica el cometido de los servicios de ficheros, sus semanticas de compartición, los tipos de servidores existentes (con estado y sin estado), los modelos de acceso existentes para acceder a un fichero así como el uso de Caché . Para terminar se exponen los conceptos de fiabilidad y replicación para concienciar de la importancia de estos de cara a realizar el diseño del sistema de un ficheros distribuido.

4.1 Introducción

Un sistema de ficheros distribuido, SFD, es un sistema en el que la ubicación de los ficheros no es relevante ya que pueden ser usados a través de la red. En dicho sistema los ficheros pueden estar en varias máquinas y pueden ser usados por un usuario en una única máquina a través de la red.

Cada persona usa varios ordenadores de manera transparente y todos se mantienen físicamente conectados en red.

El sistema de ficheros se mantiene en la red y el uso de sus ficheros no depende de ninguna ubicación dentro de la misma red del sistema, se usa por tanto ruta lógica no física.

Cuando un usuario pide un fichero para lectura o bien para escritura lo que hace es enviar al *kernel* la petición y este se encarga de interactuar mediante algún protocolo con el sistema de ficheros distribuido para localizar el fichero pedido y servírselo al usuario o aplicación que lo pidió. Por ejemplo el *kernel* de una máquina recibe la petición de un fichero y este fichero lo tiene el sistema de ficheros de otra máquina que forma parte de su sistema de ficheros distribuido aunque físicamente estén separadas.

Los sistemas de ficheros distribuidos son como se puede ver bastante más cómodos de manejar para el usuario aunque pueden tener problemas tales como el uso de algún algoritmo específico para mantener la coherencia entre los ficheros de las distintas máquinas en las que se encuentre el sistema de ficheros distribuido.

A través de un sistema de ficheros distribuido se comparte información, aumenta la comodidad y se disminuye el gasto.

4.2 Diseño

Un sistema de ficheros distribuidos se rige mediante una arquitectura cliente-servidor, con los módulos clientes ofreciendo la interfaz de acceso a los datos y los servidores encargándose del nombrado y acceso a los ficheros, diferenciándose dos componentes, el *servicio de ficheros* que maneja un sistema de ficheros plano y el servicio de directorios que traduce el nombre de ficheros que utiliza el usuario a nombres internos. La figura esquematiza la arquitectura típica de un SFD.

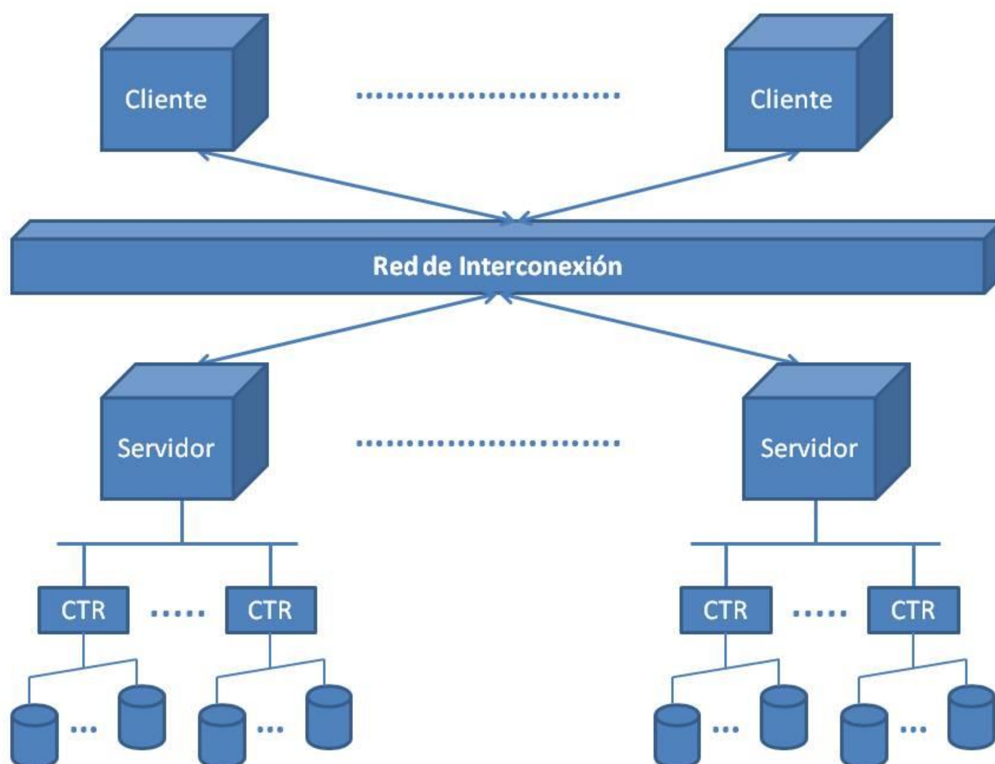

Figura 4.1 Arquitectura de un sistema de archivos distribuidos

4.3 Terminología

Un *fichero* es una abstracción que referencia a un almacenamiento permanente. Un *sistema de ficheros* es el componente responsable de organización, almacenamiento, recuperación, nombrado, compartición y protección de los ficheros en un sistema.

Un directorio es un tipo de fichero especial que provee correspondencia entre nombres y ficheros.

Un *sistema de ficheros distribuido* gestiona los dispositivos de diferentes nodos ofreciendo a los usuarios la misma visión que un *sistema de ficheros* centralizado (espacio de nombres único).

Un sistema de ficheros distribuidos tiene como principales características:

- Sistema de ficheros para un sistema distribuido, de manera que se pueda acceder a los ficheros del sistema distribuido.
- Gestiona los distintos dispositivos de diferentes nodos ofreciendo a los usuarios la misma visión que un sistema de ficheros centralizado.
- Permite que los usuarios compartan información de forma transparente.
- Misma visión desde cualquier máquina (espacio de nombres único).

4.4 Componentes de un SFD

Los distintos componentes de un sistema de ficheros distribuido son:

- Servicio de ficheros: proporciona las operaciones sobre los ficheros del sistema distribuido.
- Servicio de directorios: traduce los nombres de fichero que utiliza el usuario a nombres internos.
- Módulo cliente: permite la comunicación con el servidor de forma transparente al usuario.

4.4.1 Servicios de un SFD

Son las distintas operaciones proporcionadas por parte del sistema de ficheros distribuido a los clientes. Los servicios más importantes proporcionados por un sistema de ficheros distribuido son:

- Servicio de ficheros
- Servicio de directorios

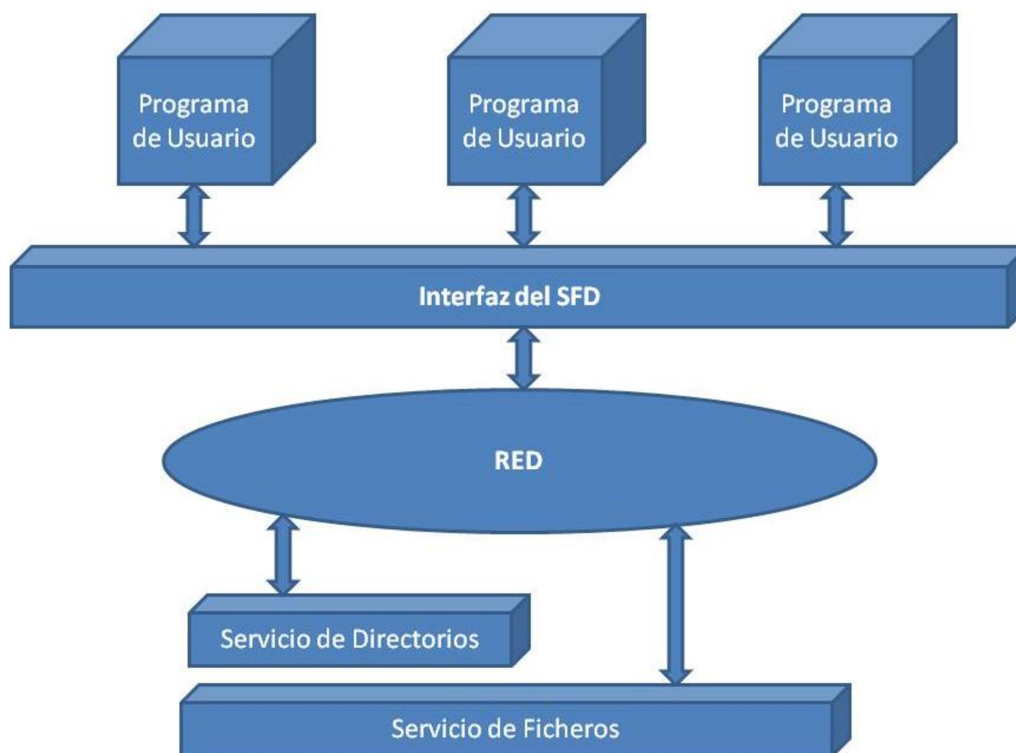

Figura 4.2 Esquema de los distintos servicios de un SFD

4.4.2 Servicio de directorios

Se encarga de la traducción de nombres de usuario a nombres internos ofreciendo una visión única del sistema de ficheros bajo una misma jerarquía de directorios para todos los usuarios.

4.4.2.1 Esquema de Nombres

El esquema de nombres permite a los usuarios asignar nombres a objetos y usar estos nombres para referirse y acceder a dichos objetos. Implica, por lo tanto, la existencia de mecanismos para localizar un objeto a partir de su nombre.

4.4.2.1.1 Propiedades

Las principales propiedades que debe cumplir un esquema de nombres son:

- Espacio de nombres global y único: De manera que todos los usuarios vean un mismo espacio de nombres, proporcionando un servicio uniforme para todos los ficheros, con total transparencia en el acceso a los nombres de los mismos.
- Transparencia de la posición: El nombre de un objeto no debe permitir inferir el lugar donde está almacenado.
- Independencia de la posición: El nombre de un objeto no necesita modificarse cuando el objeto migra. Es una propiedad más exigente que la transparencia.
- Esquema uniforme para todos los objetos: Es conveniente un esquema de nombres único para todos los objetos. Así, por ejemplo, no debería existir un servicio de nombres para los ficheros y otro para los procesos.
- El sistema debe ser escalable, Scalability: La generación y resolución de nombres no debe realizarse de forma centralizada.
- Soporte para replicación: El esquema de nombres debe permitir traducciones 1 a N.
- Nombres orientados al usuario, user-friend: Debe ofrecer al usuario un modelo de nombres flexible y adecuado para sus necesidades.

4.4.2.1.2 Niveles de traducción

La mayoría de los sistemas presentan un esquema con dos niveles:

- Nombres utilizados por los usuarios.
- Nombres internos o del sistema.

Los directorios proporcionan una asociación entre estos dos nombres.

Para satisfacer la propiedad de independencia de la posición, existe a su vez un esquema de traducción con dos niveles:

- Nivel 1: Correspondencia entre nombres de usuario y nombres internos, realizado por el servicio de directorios.
- Nivel 2: Correspondencia entre nombres internos y localización del objeto. Esta etapa es denominada Servicio de Localización, forma parte del servicio de ficheros.

4.4.2.2 Nombres de usuario

Generalmente son nombres jerárquicos representados como rutas de datos con cadenas de caracteres, *pathnames* como en UNIX. Existen tres alternativas para su creación:

- Nombres que incluyen el identificador del nodo que contiene el fichero (nodo-fichero): por lo que no proporcionan transparencia ni independencia de la posición.
- Esquema basado en montaje de directorios remotos en directorios locales: cada máquina tiene un espacio de nombres diferente. Usado en NFS.
- Espacio de nombres único: proporciona transparencia de la posición, y está basado en una generalización del mecanismo de montaje. Existen distintas posibilidades:
    - Mantener idénticas tablas de montaje de todos los nodos.
    - Incluir los puntos de montaje en los propios datos.

4.4.2.3 Nombres internos

Nombre que usa el sistema para referirse a los objetos. Debe existir un único nombre para cada objeto, identificador único o UID. El mecanismo de localización debe proporcionar independencia de la posición. Los UID pueden ser:

- Nombres no estructurados: facilitan la independencia de la posición pero dificultan la generación y resolución de nombres distribuida.
- Nombres estructurados: espacio dividido en dominios. La generación y resolución de nombres es de la siguiente manera: UID = [ID Dominio + UID Local].

4.4.2.4 Traducción de nombres de usuario

Generalmente los directorios se almacenan como ficheros, y cada entrada establece una correspondencia entre un nombre y un UID, correspondiente a cualquier tipo de objeto.

La traducción está basada en una búsqueda recursiva que puede atravesar varios dominios y, por lo tanto, implicar a varios nodos. Existen cuatro esquemas básicos de traducción.

- Transitivo: Los nodos implicados contactan entre sí para llevar a cabo la resolución. El último implicado devuelve la traducción al cliente. Dificulta el tratamiento de fallos, y no es adecuado su uso mediante RPC.

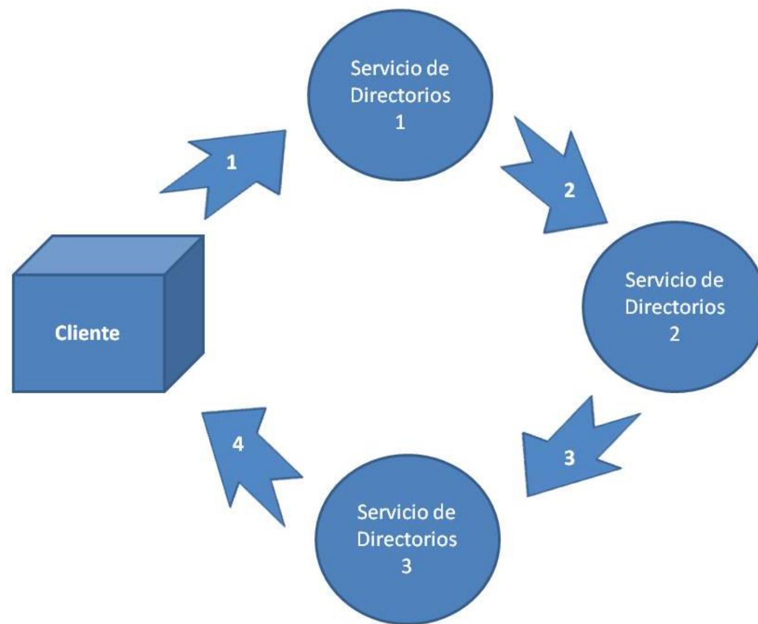

Figura 4.3 Esquema de una traducción transitiva

- Recursivo: El último nodo implicado devuelve la traducción al anterior y así sucesivamente hasta que el primer nodo la devuelve al cliente.

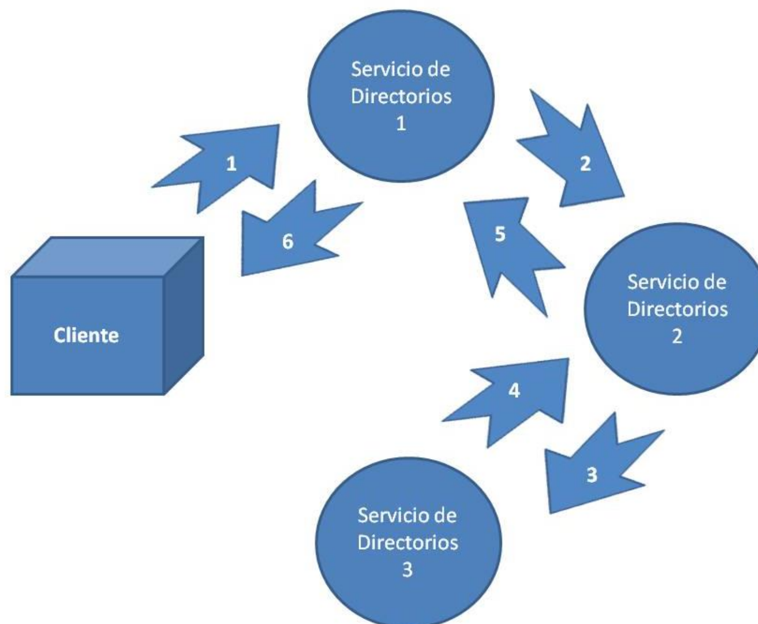

Figura 4.4 Esquema de una traducción recursiva

- Iterativo: El cliente va contactando con cada nodo implicado. Cada nodo realiza la traducción hasta que termina o alcanza un objeto fuera de su dominio devolviendo el resultado al cliente que, en caso necesario, contacta con el nodo correspondiente. Favorece el uso de mecanismos de caché de traducciones. Usado por Sprite.

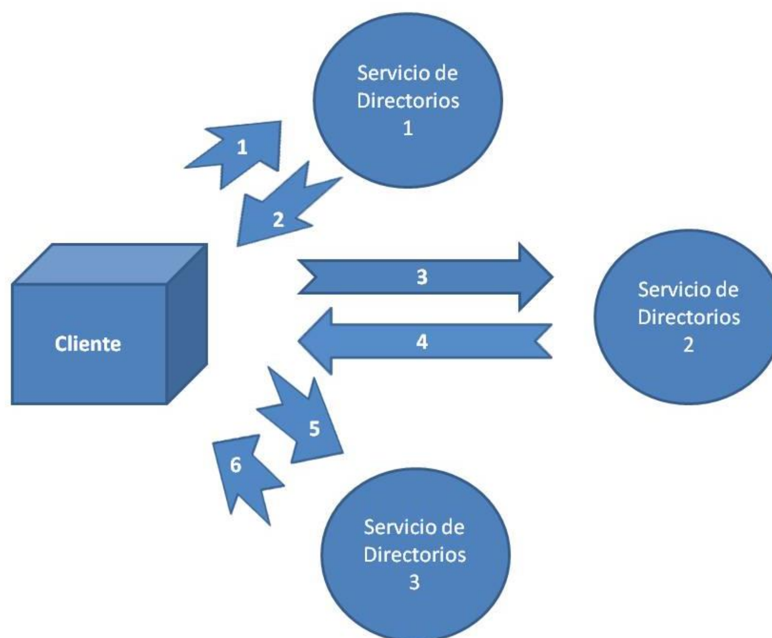

Figura 4.5 Esquema de una traducción iterativa

Dirigido por el cliente: El cliente realiza toda la traducción. Usa el servicio de directorios plano para acceder a los datos del directorio. Favorece aún más el uso de mecanismos de caché de traducciones.

Para conseguir un servicio de directorio eficiente y mejorar la capacidad de crecimiento del sistema, la mayoría de los sistemas incorporan un mecanismo de *caché de traducciones* que permite traducir un nombre sin necesidad de acceder al contenido del directorio.

4.4.3 Servicio de ficheros

Se encarga de la gestión de los ficheros y del acceso a los datos. Los aspectos relacionados con el servicio son:

- Semántica de coutilización.
- Uso de los ficheros.
- Servidores con / sin estado.
- Modelos de acceso.
- Caché de bloques y el problema de la coherencia de caché , Caching.

4.4.3.1 Semántica de compartición

Determina la visión que proporciona el sistema de ficheros a los procesos que acceden de forma concurrente a un fichero. Las semánticas estrictas son difíciles de implementar eficientemente en un sistema distribuido. Hay distintas alternativas:

- Semántica UNIX:
    - Una lectura ve los efectos de todas las escrituras previas.
    - El efecto de varias escrituras sucesivas es el de la última.
    - Los procesos pueden compartir el offset, desplazamiento, dentro de un fichero

  - Una implementación eficiente es muy difícil de conseguir en sistemas distribuidos, debido a que es necesario el manejo de cerrojos.

- Semántica de sesión:

  - Los cambios en un fichero son sólo visibles por el proceso, nodo, que modificó el fichero.
  - Cuando se cierra un fichero, fin de sesión, o se realiza una orden de actualización los cambios se hacen visibles a los procesos que abran el fichero, comienzo de sesión.
  - Existen, por lo tanto, múltiples imágenes del fichero.
  - El resultado de las sesiones que modifican concurrentemente el mismo fichero es el que finalice la sesión más tarde.
  - No se comparten offsets.

- Semántica de ficheros inmutables:

  - Su contenido no puede modificarse, sólo CRÉATE y READ, y su nombre no puede reutilizarse sin ser borrado previamente.
  - La compartición es sólo de lectura facilitando caching y replicación.
  - Cuando se quiere modificar un fichero se crea una nueva versión. Hay algunos problemas que resolver:

    - Dos procesos intentando crear una nueva versión simultáneamente.
    - Un proceso crea una nueva versión mientras otros estaban leyendo la antigua.

- Semántica de transacciones:

  - Cada proceso especifica un conjunto de operaciones sobre uno o más ficheros que forman una transacción.
  - El sistema asegura que la transacción se ejecutará de forma atómica, todo o nada.
  - También asegura que el efecto de ejecutar dos transacciones concurrentemente es equivalente a ejecutarlas secuencialmente en algún orden.

- Semántica controlada por la aplicación:

  - El sistema de ficheros proporciona primitivas para que la aplicación establezca puntos de sincronización de los datos.

- Sin una semántica definida:

  - Por ejemplo NFS.

4.4.3.2 Uso de los ficheros

Es importante conocer las características y requisitos de las aplicaciones en distintos entornos para que sirvan de guía en el diseño e implementación del sistema de ficheros. A continuación se presentan algunos datos obtenidos para tres tipos de entornos:

- Entorno UNIX (Ingeniería / oficina)
    - La mayoría de los ficheros son pequeños (<10KB).
    - Mayor frecuencia de operaciones de lectura que de escritura.
    - Generalmente los accesos son secuenciales.
    - Muchos ficheros tienen una vida muy corta,
    - La compartición de ficheros es muy poco frecuente.
- Entorno de aplicaciones paralelas de un carácter científico
    - Acceso secuencial.
    - Localidad de referencia,
    - Accesos grandes (de decenas de KB hasta MB)
    - Tasa de transferencia de datos es el parámetro más importante (del orden de 100 MB/S)
- Entorno de aplicación de bases de datos de altas prestaciones
    - Acceso aleatorio
    - Accesos pequeños (del orden de 200 bytes).
    - Número de operaciones procesadas por unidad de tiempo es el parámetro más importante (del orden de 10.000/S).

4.4.3.3 Tipos de servidores

Existen dos tipos fundamentales de servidores, los servidores con estado y sin estado.

Un servidor con estado mantiene información de las peticiones de los clientes. Cuando se abre un fichero, el servidor almacena información y da al cliente un identificador único a utilizar en las posteriores peticiones. Cuando se cierra un fichero se libera la información.

- Ventajas de los servidores con estado
    - Mensajes de petición más cortos
    - Mejor rendimiento (se mantiene información en memoria)
    - Facilita la lectura adelantada. El servidor puede analizar el patrón de accesos que realiza cada cliente.
    - Es necesario en invalidaciones iniciadas por el servidor.

- Desventajas de los servidores con estado
    - Menos tolerante a fallos ya que si el servidor cae, los estados de las peticiones que tenían los distintos clientes en el servidor se pierden.

Un *servidor sin estado* no mantiene ninguna información de las peticiones de los clientes. Las peticiones son autocontenidas, es decir la petición incorpora toda la información que necesita el servidor para acceder al fichero.

- Ventajas de los servidores sin estado
    - Más tolerante a fallos. No son necesarios open y close.
    - Se reduce el número de mensajes.
    - No se gasta memoria en el servidor para almacenar el estado.

- Desventajas de los servidores sin estado
    - No se puede realizar operaciones con semántica de estado. Como lectura adelantada, denegación de operaciones debido a operaciones hechas con anterioridad, etc.
    - Mensajes más grandes.

4.4.3.4 Modelo de acceso

El modelo de acceso determina la forma de acceder a los datos de un fichero. Los distintos modelos son:

- Modelo carga / descarga: se caracteriza por:
    - Transferencias completas del fichero del servidor al cliente.
    - Localmente se almacena en memoria o discos locales. Necesita espacio en el cliente.
    - Normalmente utiliza semántica de sesión.
    - Eficiencia en las transferencias y por lo tanto en los accesos.
    - Fácil de implementar.
    - Llamada open con mucha latencia en ficheros grandes.
    - Múltiples copias de un fichero ya que cada cliente trabaja con una copia distinta. Mala coherencia de los datos.

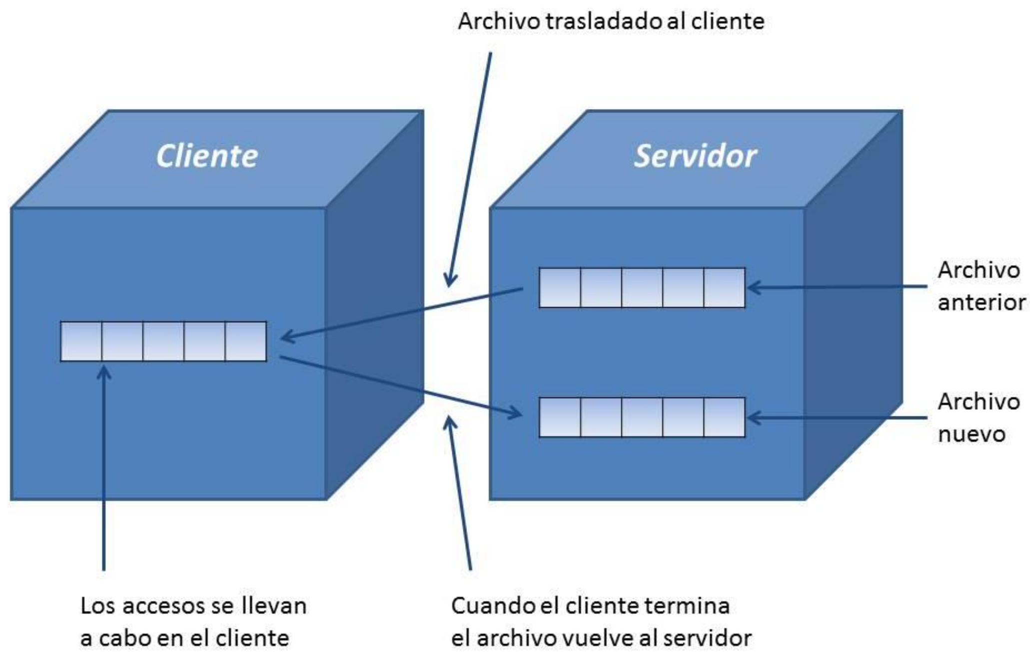

Figura 4.6 Modelos de carga / descarga

- Modelo de servicio remoto: se caracteriza por:

  - El servidor debe proporcionar todas las operaciones sobre el fichero.
  - El acceso se realiza por bloques.
  - Ofrece un modelo de comunicación tipo cliente / servidor.
  - El rendimiento no es bueno ya que los accesos se realizan a través de la red.

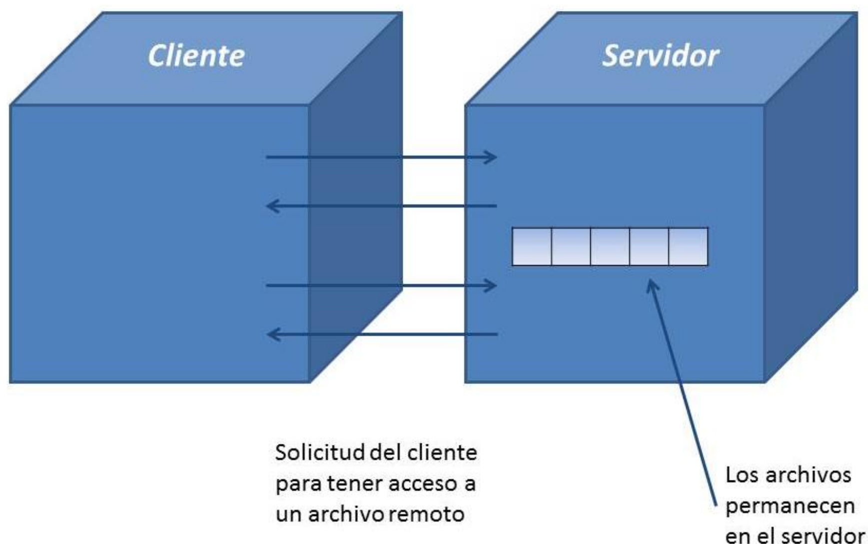

Figura 4.7 Modelos de acceso remoto

- El empleo de caché en el cliente, combina los dos modelos anteriores. Los clientes del sistema de archivos tienen una caché donde almacenan los últimos bloques solicitados.

4.4.3.5 Caching

Una *caché* o almacenamiento intermedio permite almacenar copias de los datos recientemente referenciados de tal manera que resulta mucho más rápido consultar en la *caché* que en el dispositivo en el que se almacenan los datos originariamente.

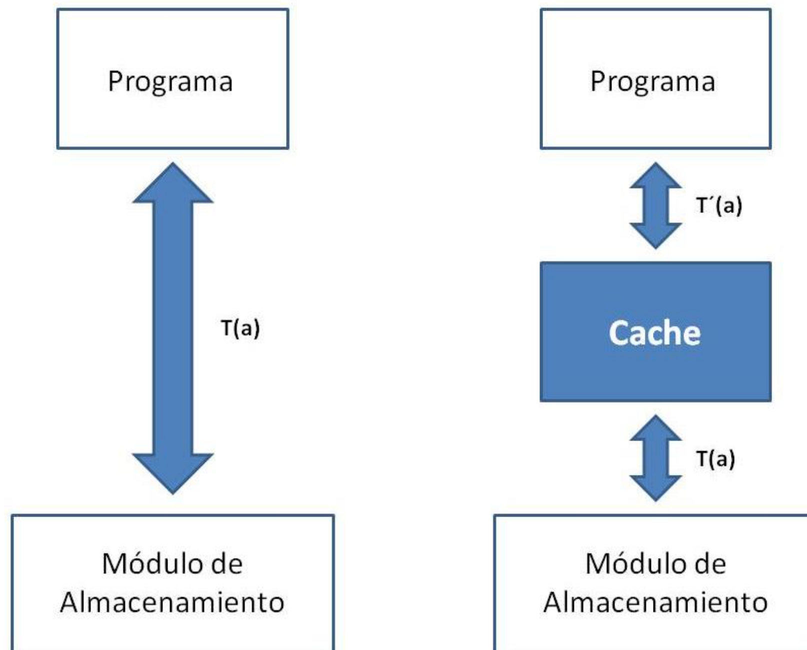

Figura 4.8 Comparativa de un sistema sin caché y otro con caché

La función T(a) representa el tiempo de acceso a un objeto a en el módulo de almacenamiento. La función T'(a) representa el tiempo de acceso a un objeto en un sistema con caché. La utilización de caché permitirá, en general, que T'(a) < T(a) para un amplio porcentaje de todos los objetos, porcentaje que dependerá de las veces que un objeto se encuentra en la caché cuando éste es referenciado. En un sistema con caché, el tiempo medio de acceso, Tm(a), a un objeto viene dado por:

$$T_m(a) = hT_c(a) + (l-h)T(a)$$

Donde h representa la tasa de aciertos de la caché, Tc(a) el tiempo de acceso a un objeto situado en la caché y T(a) el tiempo de acceso a un objeto situado en el módulo de almacenamiento.

Este mecanismo se usa de forma generalizada en los distintos niveles del Sistema de Ficheros para mejorar el rendimiento del mismo bajo tres formas:

1. La *caché* facilita los dos tipos de proximidad de referencias: *la proximidad temporal* se refiere a un acceso más rápido a aquellos bloques que son referenciados más de una vez, y *la proximidad espacial* alude a que se facilita el acceso a aquellos datos cercanos a anteriores referencias (de un mismo bloque).
2. Se pueden realizar lecturas adelantadas o *prefetching* de bloques antes que estos sean solicitados por las aplicaciones. Es beneficioso sobre todo para accesos secuenciales, ya que

permite solapar E/S con el tiempo de procesamiento de las aplicaciones. Es necesario determinar cuándo, cuanto y qué se debe leer por adelantado. El Sistema de Ficheros puede realizar esta predicción analizando los patrones de acceso del proceso o permitiendo que la aplicación pueda informarle de su comportamiento.

3. Mejora el rendimiento de las escrituras utilizando políticas de escrituras diferidas o retardada.

Se puede implantar *caché* en dos niveles, en los nodos clientes y en la memoria del servidor.

La *caché* en la *memoria del servidor* permite disminuir los accesos a disco, y no existe problema de coherencia.

En el caso de utilizar *caché* en los *nodos clientes* se reduce el número de accesos a la red y se plantea el problema de la coherencia, por lo que es necesario el empleo de un protocolo de coherencia. La *caché* puede figurar tanto en disco (gran capacidad, no volátil y no permite nodos sin disco) como en memoria, sin descartar la posibilidad de realizar las dos posibilidades de forma conjunta.

La *granularidad de la caché* indica el tamaño de los datos almacenados en la misma, variando entre 8 y 64 KB. A mayor granularidad mayor probabilidad de aciertos en los sucesivos accesos, pero también aumenta la latencia de las operaciones de E/S.

El *tamaño de la caché* se determina por los patrones de E/S que exhiben las aplicaciones e incide directamente sobre la tasa de aciertos de la misma. Si el acceso a los datos se realiza normalmente de manera secuencial, es conveniente mantener un tamaño de *caché* pequeño, ya que la posibilidad de acceso a un mismo bloque de manera casi contigua es casi nula. En cambio, si se reutilizan bloques frecuentemente puede ser conveniente el empleo de una *caché* de mayor tamaño.

Otro aspecto a tener en cuenta en el diseño de la *caché* es la *política de reemplazo.* Inicialmente la *caché* se encuentra totalmente vacía pero a medida que se van realizando los accesos se va ocupando espacio hasta que se llena. En este momento hay que seguir una política para ir sustituyendo antiguos bloques por las nuevas referencias. Existen varias políticas de reemplazo, aunque la más utilizada en sistemas distribuidos es la LRU, *Least Recently Used,* que expulsa de la *caché* aquel bloque que menos es utilizado.

Desde el punto de vista de la *actualización de los bloques* que han sido modificados, se pueden emplear, entre otras, dos políticas de actualización:

- Escritura inmediata, write through: los bloques son escritos a disco nada más ser modificados. Mejora la fiabilidad del sistema aunque reduce el rendimiento al aumentar el tiempo de ejecución de las operaciones de escritura.
- Escritura diferida, delayed-wríte: los bloques no son escritos directamente, sino que permanecen en la caché hasta que expulse el bloque. Al contrario que el de escritura inmediata, decrementa la fiabilidad debido a que los datos durante un período de tiempo son almacenados en memoria secundaria con el consiguiente riesgo de pérdida en caso de fallo. Aunque esta política mejora el rendimiento de las operaciones de escritura.

4.5 Fiabilidad

Los sistemas distribuidos pueden, en principio, presentar mayor fiabilidad que los sistemas centralizados. Sin embargo, en muchos casos la disponibilidad de un servicio, fracción del tiempo en el que está disponible, depende del funcionamiento conjunto de varios recursos.

Los distintos elementos presentes en un SFD (procesadores, red de comunicación, dispositivos de almacenamiento, *software* del SFD) pueden presentar distintos tipos de fallos (transitorios, intermitentes o permanentes) con diferentes semánticas (por ejemplo, los procesadores pueden tener semántica de fallos *fail-stop* o *bizantina).*

Es necesario realizar un diseño del SFD teniendo en cuenta este aspecto y usar técnicas de tolerancia a fallos, como la redundancia, para lograr aumentar la disponibilidad del sistema, aunque sea funcionando en un modo degradado o particionado.

4.6 Replicación

Los sistemas de ficheros distribuidos proporcionan, frecuentemente, replicación de ficheros como servicio a sus clientes, es decir, disponen de varias copias de algunos ficheros, donde cada copia está en un servidor de ficheros independientes. Este servicio ofrece una mayor tolerancia a fallos al disponer respaldos independientes de cada fichero para que en el caso de la caída de algún servidor no se pierdan los datos. De esta manera se reparte la carga de trabajo entre varios servidores favoreciendo la escalabilidad de sistema.

Un aspecto importante a destacar, es que la replicación se debe llevar a cabo de tal manera que sea transparente para el usuario.

A la hora de implantar replicación en un sistema de ficheros distribuidos, tenemos varias opciones que se detallan a continuación:

- Réplica explícita de ficheros: el programador controla todo el proceso. Cuando se genera un fichero en un determinado servidor se realizan copias adicionales en otros servidores, si así se desea. Las direcciones en la red de todas las copias se asocian con el nombre del fichero para que cuando se busque dicho fichero aparezcan todas las copias existentes dirigiéndonos a la que se encuentre disponible en ese momento.
- Réplica retrasada: se genera una copia de cada fichero en un servidor. Posteriormente se generan copias del fichero en otros servidores automáticamente, sin la intervención del programador. El sistema se encarga de recuperar una de esas copias en caso de necesidad.
- Réplica de ficheros mediante un grupo: todas las copias se crean al mismo tiempo en todos los servidores, y todas las sucesivas modificaciones son realizadas simultáneamente en todos los servidores.

La replicación trae como consecuencia la necesidad de la existencia de un protocolo de actualización de los ficheros ante modificaciones. Destacamos dos algoritmos para la resolución de este problema:

- Replicación de copia primaria: un servidor se declara como primario, siendo el resto secundarios. Ante la actualización de un fichero se realiza siempre sobre el servidor primario de forma local, encargándose éste de enviar los comandos necesarios a los secundarios para ordenar la actualización. Las lecturas son indiferentes realizarlas sobre un primario que sobre un secundario.
- Replicación mediante voto: se exige a los clientes que adquieran el permiso de varios servidores antes de leer o escribir un fichero duplicado. Para que un cliente en un esquema con N servidores tenga los permisos de escritura sobre un determinado fichero, ha de obtener la validación de al menos N/2 + 1 servidores. En caso afirmativo, el fichero es modificado y se le asocia un nuevo número de versión.

Si se quiere realizar una lectura de un fichero replicado, el cliente también debe contactar con la mitad de los servidores, más 1, y solicitarles el número de versión del fichero. En el caso de que coincidan los números de versión, esta debe ser la más reciente, debido a como es planteada la escritura.

5 Sistema de ficheros en red NFS

En esta sección se va a describir los servicios de los protocolos NFS y MOUNT.

Comienza realizando una intruducción al sistema de ficheros en red de SUN, NFS, continuamos con su definición en la que se exponen las diferentes versiones existentes, para centrarnos en la versión 2 que se ha utilizado para la implementación del proyecto.

La definición del sistema de ficheros en red engloba el modelo del sistema fisico, la autenticación, las constantes del servicio, las estructuras XDR, los tipos de datos básicos y los precodimentos del servidor. Tambíen Se exponen los problemas de implementación de NFS como son la relacación en tre cliente y servidor, la interpretación del pathname o de permisos.

Por último se aborda el montado, protocolo MOUNT, cuya definición contiene una introducción al protocolo y continua expecificando las constantes del servicio, el tamaño de las estructuras XDR, los tipos de datos básicos así como los procedimientos del servidor.

5.1 Introducción

El sistema de ficheros en red de SUN, Sun Network Filesystem o NFS, proporciona acceso transparente a sistemas de ficheros compartidos sobre redes de área local. El protocolo de NFS está diseñado para ser independiente del sistema operativo utilizado, la arquitectura de red o del protocolo de transporte utilizado. Esta independencia se obtiene mediante el uso de llamadas a procedimientos remotos, *Remote Procedure Call* o RPC.

Se encuentra implementado para gran variedad de sistemas, desde ordenadores personales hasta supercomputadores.

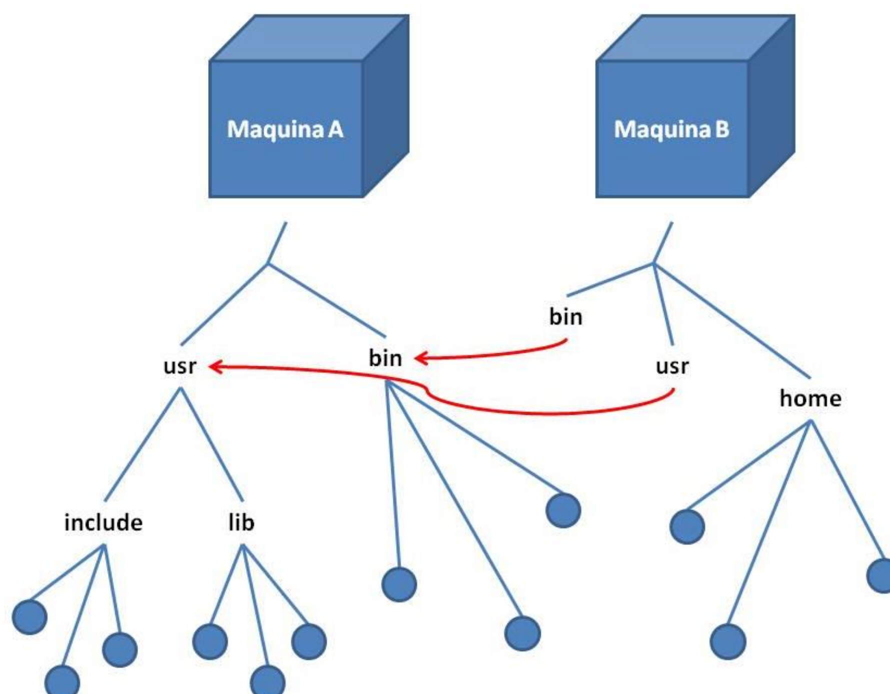

Figura 5.1 Ejemplo de asociación de directorios

El protocolo de montado permite que el servidor controle el acceso remoto a un conjunto de clientes. Realiza las funciones específicas del sistema operativo que permiten, por ejemplo, asociar árboles de directorios remotos a un cierto sistema de ficheros local.

El servidor NFS es un servidor sin estado, es decir, un servidor que no mantiene ninguna información adicional del estado sobre cualquiera de sus clientes para funcionar correctamente. Los servidores sin estado tienen una ventaja importante sobre los servidores con estado en caso de error. Con los servidores sin estado, un cliente solo necesita saber que el servidor ha caído o que la red está caída temporalmente.

El cliente de un servidor con estado, por otra parte, necesita detectar una caída del servidor y reconstruir el estado del servidor cuando vuelve a funcionar.

5.2 Definición del protocolo NFS

Existen cuatro versiones del protocolo NFS: Protocolo 2, 3 y 4. La versión 1 existía, pero era únicamente un prototipo.

A pesar de que la versión 2 de NFS fue implementada satisfactoriamente sobre varios sistemas de ficheros y sistemas operativos los ingenieros NFS de SUN recibieron muchas propuestas a mejorar en una versión superior del protocolo, tales como aumentar los atributos de los ficheros, aumentar los tipos de ficheros, incluir nuevos tipos de ficheros, etc.

En la nueva versión se eliminaron y modificaron procedimientos de la versión 2, e insertaron procedimientos nuevos.

El tamaño del manejador sufrió un incremento respecto a la versión anterior, de 32 bytes a 64 bytes. En lo referente al tamaño máximo de los datos no se refleja ningún tope para los datos, clientes y servidores pueden transferir un tamaño sin límite bajo mutuo acuerdo.

Se describe la **versión 2** del protocolo, la que se ha utilizado para realizar este proyecto.

5.2.1 Modelo del sistema de ficheros

NFS asume un sistema de ficheros jerárquico. Cada entrada en un directorio (fichero, directorio, dispositivo, etc.) tiene un nombre. Diversos sistemas operativos pueden tener restricciones en la profundidad del árbol o de los nombres usados, así como usar diversas sintaxis para representar la ruta de un fichero o de un directorio. Un "sistema de ficheros" es un árbol en un sólo servidor, generalmente un sólo disco o una partición física, con una "raíz especificada".

Algunos sistemas operativos proporcionan una operación de montado para hacer que todos los sistemas de ficheros aparezcan como uno solo árbol de directorios, mientras que otros mantienen el árbol de sistemas de ficheros, de manera que existan varios sistemas de ficheros en uno. NFS busca un componente con una ruta en un momento dado.

Hay varias razones para no hacer esto. Los ficheros y directorios son objetos similares para el sistema operativo, pero se usan procedimientos diferentes para leer directorios y ficheros.

La figura siguiente muestra la arquitectura de NFS. Es una arquitectura a tres niveles:

- Interfaz de llamadas al sistema de ficheros (POSIX,...)
- Sistema de ficheros virtual (VFS)
    - Almacena una entrada por cada archivo abierto (vnode).
    - Cada vnode apunta a un nodo-i local o a uno remoto (rnode).
    - Redirige la petición a la capa inferior correspondiente (al sistema de ficheros local o al servicio NFS).
- Servicio NFS
    - Implementa el protocolo NFS.
    - Cada rnode contiene el manejador del fichero remoto correspondiente.

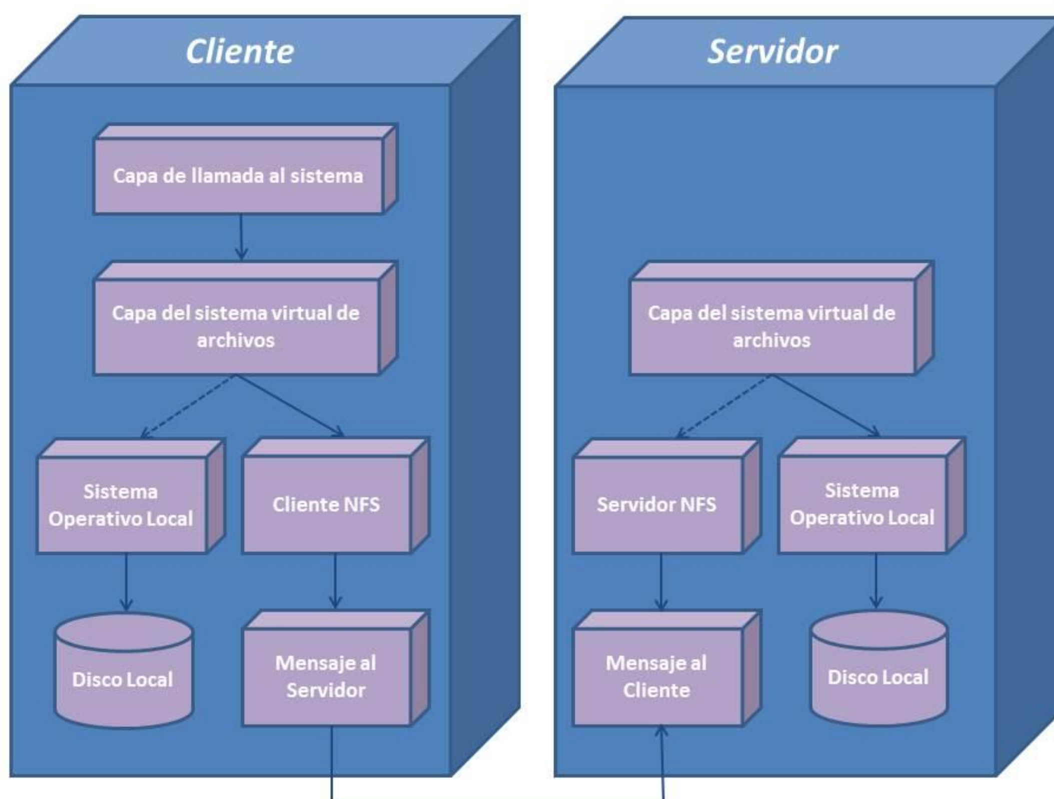

Figura 5.2 Modelo de NFS

5.2.2 Autenticación en RPC

El servicio de NFS usa *AUTHUNIX*, *AUTHDES*, o *AUTHSHORT* como estilo de autenticación, excepto el procedimiento NULL dónde *AUTHNONE* también se permite. El protocolo de transporte de NFS sólo se apoya en UDP actualmente en Linux.

El número del puerto que usa el protocolo de NFS actualmente para el protocolo de transporte UDP es 2049. Éste no es un puerto oficialmente asignado, versiones posteriores del protocolo usarán las facilidades del portmapper.

5.2.3 Constantes del servicio

Se definen las constantes necesarias para el servicio NFS:

```
// Número de programa del protocolo nfs
#define MI_NFS_PROGRAM      100003
#define MI_NFS_PORT         2049 // Puerto UDP
#define MI_NFS_VERS         2 // Número de versión
#define MI_NFS_PROC         17// Número de procedimientos

// Números de los procedimientos:
#define nfs_nulo        0// mismo que NFSPROC_NULL
#define nfs_getattr     1// mismo que NFSPROC_GETATTR
#define nfs_setattr     2// mismo que NFSPROC_SETATTR
#define nfs_root        3// mismo que NFSPROC_ROOT
#define nfs_lookup      4// mismo que NFSPROC_LOOKUP
#define nfs_readlink    5// mismo que NFSPROC_READLINK
#define nfs_read        6// mismo que NFSPROC_READ
#define nfs_writecache  7// mismo que NFSPROC_WRITECACHE
#define nfs_write       8// mismo que NFSPROC_WRITE
#define nfs_create      9// mismo que NFSPROC_CREATE
#define nfs_remove      10// mismo que NFSPROC_REMOVE
#define nfs_rename      11// mismo que NFSPROC_RENAME
#define nfs_link        12// mismo que NFSPROC_LINK
#define nfs_symlink     13// mismo que NFSPROC_SYMLINK
#define nfs_mkdir       14// mismo que NFSPROC_MKDIR
#define nfs_rmdir       15// mismo que NFSPROC_RMDIR
#define nfs_statfs      16// mismo que NFSPROC_STATFS
```

5.2.4 Tamaños de las estructuras de XDR

Estos son los tamaños, dados en número de bytes en decimal, de varias estructuras XDR usadas en el protocolo:

```
/* El máximo número de bytes que se pueden leer /
escribir en una llamada RPC */
#define MAXDATA             8192
/* Máximo número de bytes puestos en un argumento de
tipo path */
#define NFSMAXPATHLEN       1024
//El máximo de bytes que puede tener el nombre de un fichero
#define NFSMAXNAMLEN        255
// El tamaño en bytes del manejador de fichero
#define FHSIZE              32
```

5.2.5 Tipos de datos básicos

Las definiciones de XDR siguientes son las estructuras básicas y tipos usados en otras estructuras descritas más adelante.

5.2.5.1 nfs_errtbl <-> stat

```
static struct {
      int stat;
      int errno;
} nfs_errtbl[] = {
      { NFS_OK,             0                 },
      { NFSERR_PERM,        EPERM             },
      { NFSERR_NOENT,       ENOENT            },
      { NFSERR_IO,          NFSERR_IO         },
      { NFSERR_NXIO,        ENXIO             },
/*    { NFSERR_EAGAIN,      EAGAIN            }, */
      { NFSERR_ACCES,       EACCES            },
      { NFSERR_EXIST,       EEXIST            },
      { NFSERR_XDEV,        EXDEV             },
      { NFSERR_NODEV,       ENODEV            },
      { NFSERR_NOTDIR,      ENOTDIR           },
      { NFSERR_ISDIR,       EISDIR            },
      { NFSERR_INVAL,       EINVAL            },
      { NFSERR_FBIG,        EFBIG             },
      { NFSERR_NOSPC,       ENOSPC            },
      { NFSERR_ROFS,        EROFS             },
      { NFSERR_MLINK,       EMLINK            },
      { NFSERR_NAMETOOLONG, ENAMETOOLONG      },
      { NFSERR_NOTEMPTY,    ENOTEMPTY         },
      { NFSERR_DQUOT,       EDQUOT            },
      { NFSERR_STALE,       ESTALE            },
      { NFSERR_REMOTE,      EREMOTE           },
#ifdef EWFLUSH
      { NFSERR_WFLUSH,      EWFLUSH           },
#endif
      { NFSERR_BADHANDLE,   EBADHANDLE        },
      { NFSERR_NOT_SYNC,    ENOTSYNC          },
      { NFSERR_BAD_COOKIE,  EBADCOOKIE        },
      { NFSERR_NOTSUPP,     ENOTSUPP          },
      { NFSERR_TOOSMALL,    ETOOSMALL         },
      { NFSERR_SERVERFAULT, ESERVERFAULT      },
      { NFSERR_BADTYPE,     EBADTYPE          },
      { NFSERR_JUKEBOX,     EJUKEBOX          },
      { -1,                 EIO               }
};
```

Un valor de NFSOK indica que la llamada se completó con éxito. Los demás valores indican que alguna clase de error ocurrió en el lado del servidor durante llamada del procedimiento. Los valores del error se derivan de los códigos de error de UNIX.

En la siguiente tabla se explica con mayor detalle las constantes con sus detalles:

Nombre del error	Valor	Significado
NFS OK	0	La llamada se realizó satisfactoriamente.
NFSERR_PERM	1	No existe propietario o no tiene los permisos adecuados.
NFSERR NOENT	2	No existe tal fichero o directorio.
NFSERR_IO	5	Error de Entrada / Salida.
NFSERR_NXIO	6	Error de Entrada / Salida. No existe tal dispositivo o dirección.
NFSERR ACCES	13	Permisos denegados.
NFSERR_EXIST	17	El fichero existe.
NFSERR_XDEV	18	Inválido el enlace del dispositivo.
NFSERR_NODEV	19	No existe el dispositivo.
NFSERR_NOTDIR	20	No es un directorio.
NFSERR ISDIR	21	Es un directorio.
NFSERR_INVAL	22	Argumentos incorrectos.
NFSERR_FBIG	27	Tamaño del fichero demasiado grande.
NFSERR_NOSPC	28	No hay espacio en el dispositivo.
NFSERR ROFS	30	Sistema de ficheros de solo lectura.
NFSERR_MLINK	31	Demasiados enlaces duros.
NFSERRJSFAMETOOLONG	63	El nombre del fichero es demasiado grande.
NFSERR NOTEMPTY	66	Directorio no esta vacío.
NFSERR_DQUOT	69	Cuota excedida.
NFSERR_STALE	70	Manejador no válido.
NFSERR REMOTE	71	El manejador no es de un servidor local
NFSERR BADHANDLE	10001	El manejador NFS es ilegal.
NFSERR_NOTSYN	10002	Problemas en la modificación de los atributos.
NFSERR_BADCOOKIE	10003	Mala lectura del readdir.
NFSERR NOTSUPP	10004	Operación no soportada
NFSERRJTOOSMALL	10005	El buffer es demasiado grande.
NFSERR SERVERFAULT	10006	Error del servidor desconocido.
NFSERR BADTYPE	10007	Inválido tipo de fichero.
NFSERR_JUKEBOX	10008	Operación del servidor pendiente.

5.2.5.2 nfs_ftype

Definido en el kernel de linux, include/linux/nfs.h.

```
enum nfs_ftype {
        NFNON = 0,
        NFREG = 1,
        NFDIR = 2,
        NFBLK = 3,
        NFCHR = 4,
        NFLNK = 5,
        NFSOCK = 6,
        NFBAD = 7,
        NFFIFO = 8
};
```

El tipo de datos *nfs_ftype* es un enumerado que indica el tipo de un fichero. El tipo NFNON indica que no es un fichero, NFREG es un fichero regular, NFDIR es un directorio, NFBLK es un dispositivo especial de bloques, NFCHR es un dispositivo especial de caracteres, NFLNK es un enlace simbólico, NSOCK es un socket y NFFIFO nombre de tubería.

5.2.5.3 fhandle

```
typedef char          fhandle[FHSIZE];
```

El *fhandle* es el manejador de fichero pasado entre el servidor y el cliente. Se hacen todas las operaciones del fichero usando el manejador para hacer referencia a un fichero o a un directorio. El manejador del fichero contiene la información que el servidor necesita para distinguir un fichero individual.

Los datos de los cuales se compone un fhandle son:

Identificador del sistema de ficheros.	Número de *i-nodo* del fichero.	Número de generación del número de *i-nodo*.

El número de generación del número de i-nodo se utiliza para que un fichero que se ha creado se identifique con otro fichero que se borró. De manera que cada vez que se crea un fichero, y se le asigna un i-nodo, este incrementa su número de generación, y así poder identificar el nuevo valor del i-nodo con el antiguo.

5.2.5.4 nfs_fattr

Estructura definida en el kernel de linux, /include/linux/nfs_xdr.h.

```
struct nfs_fattr {
	unsigned short	valid;// which fields are valid
	__u64		pre_size; // pre_op_attr.size
	__u64		pre_mtime;//pre_op_attr.mtime
	__u64		pre_ctime;//pre_op_attr.ctime
	enum nfs_ftype	type;// always use NFSv2 types
	__u32		mode;
	__u32		nlink;
	__u32		uid;
	__u32		gid;
	__u64		size;
	union {
		struct {
			__u32	blocksize;
			__u32	blocks;
		} nfs2;
		struct {
			__u64	used;
		} nfs3;
	} du;
	__u32		rdev
	__u64		fsid;
	__u64		fileid;
	__u64		atime;
	__u64		mtime;
	__u64		ctime;
};
```

La estructura contiene los atributos de un fichero:

- valid: Indica que los campos son válidos.
- type: es el tipo del fichero.
- nlink: es el número de enlaces físicos al fichero, el número de diversos nombres para el mismo fichero.
- uid: es el número de identificación del propietario del fichero.
- gid: es el número de identificación del grupo al que pertenece el propietario del fichero.
- size: es el tamaño en bytes del fichero.
- blocksize: es el tamaño en bytes de un bloque del fichero.
- rdev: es el número de dispositivo del fichero si el tipo es NFCHR o NFBLK.
- blocks: es el número de bloques del fichero.
- used: espacio de disco usado. Para la versión 3.
- fsid: es el identificador del sistema de ficheros en el cual se encuentra el fichero.
- fileid: es un número que identifica únicamente el fichero dentro de su sistema de ficheros.
- atimes: la fecha de último acceso.
- mtime: es la fecha en que los datos del fichero fueron por última vez modificados.

- ctime: es la fecha en que el estado del fichero fue cambiado por último vez. Al escribir en el fichero también cambia el ctime si el tamaño del fichero cambia.
- mode: bits de permisos del fichero. Es el modo de acceso codificado como conjunto de dígitos binarios.

El tipo del fichero está especificado en los campos de modo y del tipo de fichero. Esto es realmente un fallo de funcionamiento en el protocolo y será fijado en las versiones futuras. Las descripciones dadas abajo especifican las posiciones en binario de cada uno de los campos.

Bit	Descripción
0040000	Es un directorio; el campo "type" debería ser NFDIR
0020000	Es el carácter de un fichero especial; "type" debería ser NFCHR
0060000	Es un bloque especial de un fichero; "type" debería ser NFBLK
0100000	Es un fichero normal; "type" debería ser NFREG
0120000	Es un enlace físico; "type" debería ser NFLNK
0140000	Es el nombre del socket; "type" debería ser NFNON
0004000	Tiene el identificador del usuario en ejecución
0002000	Tiene el identificador del grupo en ejecución
0001000	Salva los datos intercambiados incluso después de su utilización
0000400	Permiso de lectura para el propietario
0000200	Permiso de escritura para el propietario
0000100	Permiso de búsqueda y ejecución para el propietario
0000040	Permiso de lectura para el grupo
0000020	Permiso de escritura para el grupo
0000010	Permiso de búsqueda y ejecución para el grupo
0000004	Permiso de lectura para otros
0000002	Permiso de escritura para otros
0000001	Permiso de búsqueda y ejecución para otros

Nota: Los bits son iguales que los bits del modo que se devuelven en la llamada del sistema stat(2) en el sistema de UNIX.

5.2.5.5 iattr

Estructura definida en el kernel de linux, /include/linux/fs.h.

```
struct iattr {
        unsigned int            ia_valid;
        umode_t                 ia_mode;
        uid_t                   ia_uid;
        gid_t                   ia_gid;
        loff_t                  ia_size;
        time_t                  ia_atime;
        time_t                  ia_mtime;
```

```
        time_t                  ia_ctime;
        unsigned int            ia_attr_flags;
};
```

La estructura contiene los atributos del fichero que se pueden fijar desde el cliente. Los campos son iguales que para la estructura nfs_fattr. Un size igual a cero significa que el fichero debe ser borrado. Un valor de -1 indica un campo que debe ser ignorado.

5.2.5.6 filename

```
      typedef char            *filename;
```

filename se utiliza para definir los campos en los cuales se representa el nombre de un fichero o directorio.

5.2.5.7 path

```
      typedef char            *path;
```

El tipo de datos *path* es una ruta de datos. El servidor lo considera como una cadena sin estructura interna, pero para el cliente es el nombre de un nodo en el árbol del sistema de ficheros.

5.2.5.8 diropargs

```
      struct diropargs {
            fhandle           dir;
            filename          name;
      };
      typedef struct diropargs     diropargs;
```

diropargs es la estructura utilizada en operaciones que utilizan directorios. El manejador *dir* es el manejador del directorio en el cual encontrar el fichero *name.*

5.2.5.9 diropres

```
      struct diropok {
            fhandle                 file;
            struct nfs_fattr        attributes;
      };
      typedef struct diropok        diropok;

      struct diropres {
            unsigned int            status;
            union {
                  diropok           fhand_attr;
            } diropres_u;
      };
      typedef struct diropres       diropres;
```

Los resultados de una operación sobre un directorio se devuelven en una estructura de tipo *diropres.* Si la llamada tuvo éxito, *status* es igual a cero, se devuelve el manejador del directorio y los atributos asociados al directorio en la estructura *diropok*.

5.2.6 Procedimientos del servidor

El protocolo se define con un conjunto de procedimientos que tienen argumentos y resultados definidos usando el lenguaje RPC. Una descripción abreviada de la función de cada procedimiento debe proporcionar bastante información para permitir la puesta en práctica. Todos los procedimientos del protocolo NFS se asume que son síncronos.

Cuando se devuelve al cliente el control de una llamada a un procedimiento remoto, el cliente puede asumir que la operación ha terminado. Por ejemplo, una petición de un cliente que realice una operación de escritura, puede hacer al servidor actualizar los bloques de datos, la información de los bloques del sistema de ficheros, y la información de los atributos del fichero (el tamaño y la modificación de las fechas).

Cuando la llamada que realiza la escritura acaba y devuelve al cliente la repuesta, este puede asumir que la escritura ha sido realizada, incluso en caso de que haya habido una caída en el servidor, y se borren los datos escritos. Esta es una parte muy importante del estado del servidor. Si el servidor está esperando para vaciar los datos de peticiones remotas, el cliente tuvo que mantener esas peticiones de modo que pudiera volverlas a enviar en caso de una caída del servidor, porque es posible que el servidor aún no se haya actualizado.

A continuación se definen los procedimientos remotos proporcionados por el servicio NFS:

Definición del servicio NFS:

```
struct rpc_program    mi_nfs_program = {
      "nfs",            // Nombre del protocolo
      MI_NFS_PROGRAM,  // Número de programa.
      // Tamaño
      sizeof(mi_nfs_version) / sizeof(mi_nfs_version[0]),
      mi_nfs_version,  // Array de versiones
      &mi_nfs_rpcstat, // Estadísticas
};

struct rpc_stat mi_nfs_rpcstat = { &mi_nfs_program };
```

Array de versiones: contiene las distintas versiones que queramos definir para el servicio.

```
static struct rpc_version *mi_nfs_version[] = {
      NULL,                 // Para la versión cero
      NULL,                 // Para la versión uno
      &mi_nfs_version2,     // Para la versión dos
//#ifdef CONFIG_NFS_V3
//    &mi_nfs_version3,     // Para la versión tres
//#endif
};
```

Versión 2 de NFS:

```
struct rpc_version          mi_nfs_version2 = {
     MI_NFS_VERS,       // Número de versión de nfs
     MI_NFS_PROC,       // Número de procedimientos
     mi_nfs_procedures //Procedimientos de la versión
};
```

Definición de los procedimientos usando la macro: he definido una macro para simplificar la definición, el primer parámetro es para el nombre del procedimiento, el segundo identifica el nombre de la función xdr encargada de codificar y enviar la petición al servidor y el último la función xdr encargada de recoger y decodificar la respuesta devuelta.

```
static struct rpc_procinfo mi_nfs_procedures[17] = {
    PROC(null,        enc_void,        dec_void),
    PROC(getattr,     fhandle,         attrstat),
    PROC(setattr,     sattrargs,       attrstat),
    PROC(root,        enc_void,        dec_void),
    PROC(lookup,      diropargs,       diropres),
    PROC(readlink,    fhandle,         readlinkres),
    PROC(read,        readargs,        readres),
    PROC(writecache,  enc_void,        dec_void),
    PROC(write,       writeargs,       attrstat),
    PROC(create,      createargs,      diropres),
    PROC(remove,      diropargs,       stat),
    PROC(rename,      renameargs,      stat),
    PROC(link,        linkargs,        stat),
    PROC(symlink,     symlinkargs,     stat),
    PROC(mkdir,       createargs,      diropres),
    PROC(rmdir,       diropargs,       stat),
    PROC(statfs,      fhandle,         statfsres),
};
```

Definición de los tamaños de los tipos de datos:

```
#ifndef MAX
#define MAX(a, b)     (((a) > (b))? (a) : (b))
#endif

#define NFS_fhandle_sz    8
#define NFS_filename_sz   1 + (NFS2_MAXNAMLEN>>2)
#define NFS_fattr_sz      17

#define NFS_enc_void_sz   0
#defineNFS_diropargs_sz   NFS_fhandle_sz+NFS_filename_sz

#define NFS_dec_void_sz   0
#define NFS_diropres_sz   1+NFS_fhandle_sz+NFS_fattr_sz
```

Definición de la macro de las llamadas al servidor:

```
#define PROC(proc, argtype, restype)                    \
    { "nfs_" #proc,                                     \
      (kxdrproc_t) nfs_xdr_##argtype,                   \
      (kxdrproc_t) nfs_xdr_##restype,                   \
      MAX(NFS_##argtype##_sz,NFS_##restype##_sz) << 2, \
      0                                                 \
    }
```

5.2.6.1 Null

```
rpc_call(mi_nfs_clnt, nfs_nulo, argp, clnt_res, 0);
```

Este procedimiento no hace ninguna función. Está disponible para comprobar tiempos de respuesta y el estado del servidor.

5.2.6.2 Leer atributos

```
struct nfs_fattr            *fattr

rpc_call(mi_nfs_clnt, nfs_getattr,
fh->fhstatus_u.directory, fattr, 0);
```

Si el estado de la respuesta es NFSOK, la respuesta, *struct nfs_fattr*, de la petición contiene los atributos para el fichero dado por el manejador del fichero pasado como parámetro.

5.2.6.3 Escribir atributos

```
struct sattrargs {
      fhandle                 *file;
      struct iattr            *attributes;
};
typedef struct sattrargs    sattrargs;

struct sattrargs            arg;

rpc_call(mi_nfs_clnt, NFSPROC_SETATTR, &arg, fattr, 0);
```

El argumento de atributos contiene los campos que se van a escribir en el fichero. Si el estado de la petición es NFSOK, los atributos devueltos son los atributos del fichero después de realizar la operación *setattr.*

Nota: El un campo con valor igual a -1 indica que ese campo no se utiliza. Se cambiará en la siguiente versión del protocolo.

5.2.6.4 Obtener manejador raíz

```
rpc_call(mi_nfs_clnt, nfs_root, argp, clnt_res, 0);
```

Obsoleto. Este procedimiento no se utiliza porque obtener el manejador de fichero de la raíz de un sistema de ficheros requiere mover el nombre de las distintas rutas de datos entre el cliente y el servidor. Para hacer esto correctamente tendríamos que definir una representación estándar de las rutas de datos que se encuentran en la red. En su lugar, la función búsqueda del

manejador de fichero de raíz del sistema de ficheros se realiza mediante el procedimiento de "montado un directorio", véase la definición del protocolo del montado.

5.2.6.5 Buscar un fichero

```
diropargs        arg;
dirополнres        res;

rpc_call(mi_nfs_clnt, NFSPROC_LOOKUP, &arg, &res, 0);
```

Si la respuesta del estado es NFSOK, el resultado de la operación es el manejador del fichero y los atributos asociados al fichero pasado como parámetro. Se debe pasar el manejador del directorio donde se encuentra el fichero y el nombre del fichero.

5.2.6.6 Leer enlace simbólico

```
struct readlinkres {
      unsigned int            status;
      union {
            path              data;
      } readlinkres_u;
};
typedef struct readlinkres readlinkres;

readlinkres                   clnt_res;

rpc_call(mi_nfs_clnt,NFSPROC_READLINK, fh, &clnt_res,0);
```

Si el estado tiene el valor NFSOK, la respuesta a la operación son los datos del enlace simbólico dado por el fichero pasado como argumento su manejador.

<u>Nota</u>: puesto que NFS analiza siempre la ruta en el cliente, la ruta de un enlace simbólico puede significar algo diferente (o ser sin sentido) en un cliente distinto o en el servidor si se utiliza una sintaxis distinta de la ruta.

5.2.6.7 Leer de fichero

```
struct readargs {
      fhandle                 file;
      u_long                  offset;
      u_long                  count;
      u_long                  totalcount;
};
typedef struct readargs       readargs;

typedef struct {
      unsigned int            nfsdata_len;
      char                    *nfsdata_val;
} nfsdata;

struct datosRes {
      struct nfs_fattr        attributes;
      nfsdata                 data;
```

```
};
typedef struct datosRes     datosRes;

struct readres {
      unsigned int status;
      union {
            datosRes          fich_read;
      } readres_u;
};
typedef struct readres      readres;

readargs                    args;
readres                     res;

rpc_call(mi_nfs_clnt, nfs_read, &args, &res, 0);
```

Devuelve el número de bytes leídos del fichero, comenzando en un desplazamiento del fichero dado, empezando a contar desde el principio del fichero. El primer byte del fichero está en el desplazamiento u *offset* cero. Los atributos del fichero se devuelven en el campo *attributes*.

Nota: El argumento *totalcount* no se usa, y se quitará en la siguiente revisión del protocolo.

5.2.6.8 Escribir en caché

```
rpc_call(mi_nfs_clnt, nfs_writecache, argp, clnt_res,0);
```

Se utilizará en la siguiente versión del protocolo.

5.2.6.9 Escribir a fichero

```
struct writeargs {
      fhandle                 file;
      u_long                  beginoffset;
      u_long                  offset;
      u_long                  totalcount;
      nfsdata                 data;
};
typedef struct writeargs    writeargs;

writeargs                   args;
struct nfs_fattr            *res;

rpc_call(mi_nfs_clnt, nfs_write, &args, res, 0);
```

Escribe los datos empezando en el desplazamiento dado, el cual empieza a contar a partir del principio del archivo. El primer byte del archivo está en la posición cero del desplazamiento. Si el estado de los datos de retorno es NFSOK, entonces la petición ha sido completada. La función *write* es atómica. No se mezclarán los datos de esta llamada para escribir los datos de las llamadas posteriores de otros clientes de otro cliente.

Nota: Se ignoran los argumentos *beginoffset* y *totalcount* y se eliminaran en la próxima versión del protocolo.

5.2.6.10 Crear fichero

```
struct createargs {
      diropargs              where;
      struct iattr           attributes;
};
typedef struct createargs   createargs;

createargs                  arg;
diropres                    res;

rpc_call(mi_nfs_clnt, nfs_create, &arg, &res, 0);
```

El archivo se crea en el directorio dado por el parámetro "dir" que es el manejador del directorio. Los atributos iniciales del nuevo archivo son dados por los atributos pasados por parámetro. Si el estado de la llamada es igual a NFSOK indica que el archivo fue creado. Cualquier otro tipo de estado indica que el funcionamiento falló y ningún archivo fue creado.

Nota: Si el fichero ya existe la operación no hace nada.

5.2.6.11 Borrar fichero

```
diropargs                   args;

rpc_call(mi_nfs_clnt, nfs_remove, &args, NULL, 0);
```

El archivo borrado está determinado por el nombre del fichero y el manejador de directorio dado por el campo *dir.* Si la respuesta a es igual a NFSOK la entrada del directorio ha sido eliminada.

Nota: posibilidad de operación no idempotente.

5.2.6.12 Renombrar fichero

```
struct renameargs {
      diropargs              from;
      diropargs              to;
};
typedef struct renameargs   renameargs;

renameargs                  args;

rpc_call(mi_nfs_clnt, NFSPROC_RENAME, &args, NULL, 0);
```

El archivo existente, indicado por el campo *from.name,* que se encuentra en el directorio dado por *from.dir* se renombra al nombre que aparece en el campo *to.name* en el directorio dado por *to.dir.* Si la respuesta es NFSOK, el archivo fue renombrado. La función es atómica en el servidor.

Nota: posibilidad de operación no idempotente.

5.2.6.13 Crear enlace físico

```
struct linkargs {
      fhandle                 from;
      diropargs               to;
};
typedef struct linkargs       linkargs;

linkargs                      args;

rpc_call(mi_nfs_clnt, nfs_link, &args, NULL, 0);
```

Crea el archivo *to.name* en el directorio dado por *to.dir* que es un enlace físico al archivo existente dado por el campo *from.* Si el valor de retorno es NFSOK, el enlace fue creado. Cualquier otro valor de retorno indica un error, y el enlace no fue creado.

Un enlace físico es un enlace al i-nodo del fichero, de manera que se pueden tener varias referencias al mismo fichero.

5.2.6.14 Crear enlace simbólico

```
struct symlinkargs {
      diropargs               from;
      path                    to;
      struct iattr            attributes;
};
typedef struct symlinkargs   symlinkargs;

symlinkargs                   args;

rpc_call(mi_nfs_clnt, nfs_symlink, &args, NULL, 0);
```

Crea el archivo con el nombre que se encuentra en el campo *from.name* con el tipo *ftype* igual a NFLNK en el directorio dado por el campo *from.dir*. El nuevo archivo contiene la ruta recogida en el campo *to* y tiene los atributos iniciales dados por los *attributes*. Si el valor de retorno es NFSOK, un enlace fue creado. Cualquier otro valor del retorno indica un error, y el enlace no fue creado. Un enlace simbólico es un acceso directo a otro archivo. La ruta dada en el campo *to* no es interpretada por el servidor, lo único que hace es guardar el campo en el archivo recientemente creado.

Cuando el cliente hace referencia a un archivo que es un enlace simbólico, normalmente se reinterpretan transparentemente los volúmenes del enlace simbólico como una ruta de datos para sustituir. Con la llamada READLINK se devuelve los datos al cliente para la interpretación.

Nota: En los servidores de UNIX los atributos no se usan nunca, desde que los enlaces simbólicos tienen modo 0777.

5.2.6.15 Crear directorio

```
    createargs              args;
    diropres                res;

rpc_call(mi_nfs_clnt, nfs_mkdir, &args, &res, 0);
```

El nuevo directorio *where.name* se crea en el directorio dado por *where.dir*. Los atributos iniciales del nuevo directorio son dados por los *attributes*. Si el estado de la estructura devuelta es NFSOK indica que el nuevo directorio fue creado, y los campos *file* y *attributes* de la respuesta son su manejador del directorio y sus atributos respectivamente. Cualquier otro estado indica que el procedimiento falló y ningún directorio fue creado.

5.2.6.16 Borrar directorio

```
diropargs                   args;

rpc_call(mi_nfs_clnt, nfs_rmdir, &args, NULL, 0);
```

Borra el directorio pasado por parámetro, el campo n*ame* es el nombre del directorio y el campo *dir* el manejador del directorio donde se encuentra.

5.2.6.17 Conseguir los atributos del sistema

```
struct nfs_fsinfo           res;

rpc_call(mi_nfs_clnt, nfs_statfs, fh, &res, 0);
```

Si el estado es NFSOK, entonces la estructura *nfs_fsinfo* de la respuesta contiene los atributos para el sistema de ficheros que contiene el archivo referido a por el *fhandle* de la entrada. La estructura *nfs_fsinfo* está definida en el kernel de Linux, include/linux/nfs_xdr.h

Nota: Esta llamada no funciona bien si el sistema de ficheros tiene bloques de tamaño variable.

5.3 Problemas de implementación de NFS

El protocolo NFS se ha diseñado para ser independiente del sistema operativo, pero desde que esta versión se diseñó en un ambiente UNIX, tiene la semántica similar al sistema de ficheros de UNIX.

Esta sección expone algunos de los problemas semánticos específicos de la implementación.

5.3.1 Relación entre Servidor / Cliente

El protocolo de NFS está diseñado para permitir que los servidores sean tan simples y generales como sea posible. Esto puede hacer que el servidor no pueda contemplar semánticas complejas, como el borrado de un fichero abierto. Esta operación en los sistemas UNIX permite que un fichero sea borrado inmediatamente al cerrarlo, mientras que al ser NFS un servidor sin estado, el fichero se borrará nada más se realice la operación.

5.3.2 Interpretación del pathname

Pueden existir problemas al analizar los *pathnames* o rutas de datos en el cliente. Por ejemplo, los enlaces simbólicos podrían tener las interpretaciones diferentes en clientes distintos.

5.3.3 Problemas de permisos

Al ser NFS un servidor sin estado, no puede diferenciar entre un fichero abierto por un cliente y otro que no. La mayoría de sistemas operativos comprueban el permiso en el momento abrir un fichero y en cada llamada de lectura o escritura se comprueba que el fichero siga abierto, en cambio en los servidores sin estado, el servidor no tiene ninguna idea de que fichero está abierto, con lo que debe comprobar los permisos en cada llamada de lectura o escritura. En un sistema de ficheros local, un usuario puede abrir un fichero y cambiar los permisos para que a nadie se permita tocarlo, pero todavía podría escribir en el fichero porque está abierto. En NFS, por el contrario, la escritura fallaría. Un problema similar se tiene al compartir un fichero a través de la red. El sistema operativo normalmente verifica el permiso para ejecutar antes de abrir un archivo, y lee los bloques del fichero abierto. Una vez abierto el fichero no importa el cambiar el permiso de ejecución.

Otro problema de los permisos ocurre al acceder a un sistema de ficheros remotos siendo superusuario en sistema de ficheros local. En la mayoría de los sistemas operativos, el superusuario (con el identificador de usuario 0) tiene acceso a todos los archivos no importa qué permisos y propiedades estos tengan. Este permiso de superusuario no puede permitirse en el servidor, ya que el superusuario de otro ordenador cualquiera podría suplantar al superusuario y acceder a todos los ficheros. El servidor de UNIX por defecto, pasa el identificador de usuario O a (-2) antes de hacer la comprobación de acceso para evitar este problema.

5.4 Definición del protocolo de montado

A continuación se describe el protocolo de montado, protocolo utilizado para poder obtener el manejador origen de los directorios compartidos.

5.4.1 Introducción

El protocolo de montado es independiente del protocolo de NFS, pero está relacionado con él. Proporciona los servicios específicos del sistema operativo para conseguir los manejadores y nombres de los directorios que NFS exporta, valida la identidad del usuario, y verifica los permisos de acceso. Los clientes usan el protocolo de montado para conseguir el primer manejador del directorio el cual permite la entrada al sistema de ficheros remoto.

El servidor de montado es un servidor de estado porque el servidor mantiene una lista de las demandas de montado por cada cliente. La información de lista de montado no es crítica para el funcionamiento correcto del cliente y el servidor. Sólo se piensa para el uso de auditorías.

La versión uno del protocolo de montado se usa con versión dos del protocolo de NFS. El único punto que los une es la estructura *fhandle*, manejador de ficheros, que es la misma para ambos protocolos.

5.4.2 Información adicional del protocolo

Aquí se recoge información necesaria para la implementación del protocolo:

- Autenticación: el servicio de montado usa sólo las autenticaciones del tipo *AUTHUNIX* y *AUTHDES*.
- Protocolo de transporte utilizado: el servicio de montado funciona tanto con UDP como con TCP.
- Número del puerto: se utiliza el portmapper del servidor, para encontrar el servicio de montado.

5.4.3 Constantes del servicio

Se definen las constantes necesarias para el servicio MOUNT:

```
// Número de programa del protocolo mount
#define MI_MOUNT_PROGRAM    100005
#define MI_MOUNT_PORT       1025 // Puerto para UDP
#define MI_MOUNT_VERS       1 // Número de versión
#define MI_MOUNT_PROC       6 // Número de procedimientos
// Números de los procedimientos
#define nulo                0 // mismo que MOUNTPROC_NULL
#define montar              1 // mismo que MOUNTPROC_MNT
#define dump                2 // mismo que MOUNTPROC_DUMP
#define umnt                3 // mismo que MOUNTPROC_UMNT
#define umntall             4
// Mismo que MOUNTPROC_UMNTALL
#define exportar            5//mismo que MOUNTPROC_EXPORT
```

5.4.4 Tamaño de las estructuras XDR

Éstos son los tamaños, dados en bytes decimales, de varias de las estructuras de XDR usadas en el protocolo:

```
// Número máximo de caracteres en un path
#define MNTPATHLEN          1024
// Número máximo de caracteres en un nombre
#define MNTNAMLEN           255
// Tamaño del manejador de fichero
#define FHSIZE              32
```

5.4.5 Tipo de datos básicos

Las definiciones de XDR siguientes son las estructuras básicas y tipos usados en otras estructuras descritas más adelante.

5.4.5.1 fhandle

```
typedef char                fhandle[FHSIZE];
```

Es igual que el *fhandle* de la definición XDR de la versión 2 del protocolo de NFS, vea los Tipos de los Datos Básicos en la definición del protocolo de NFS, anteriormente descritos.

5.4.5.2 fhstatus

```
 struct fhstatus {
      u_long                status;
      union {
            fhandle         directory;
      } fhstatus_u;
};
typedef struct fhstatus     fhstatus;
```

El tipo *fhstatus* se utiliza para recoger manejadores de directorios. Si valor del campo status devuelto es igual a cero, la llamada se completó con éxito, y se consigue el manejador de fichero para el directorio. Cualquier otro valor de este campo significa que ha ocurrido un error.

5.4.5.3 dirpath

```
typedef char                *dirpath;
```

El tipo *dirpath* es una ruta de datos de un directorio del servidor.

5.4.5.4 name

```
typedef char                *name;
```

El tipo *name* es una cadena de caracteres usada para los nombres.

5.4.6 Procedimientos del servidor

A continuación se definen los procedimientos remotos proporcionados por el servicio de montado:

Tamaño del path del protocolo mount:

```
#define MNT_dirpath_sz      (1 + 256)
```

Declaración del servicio mount:

```
static struct rpc_procinfo mi_mount_procedures[6] = {
      { "mnt_null", // Nombre del procedimiento
            // xdr encode function
            (kxdrproc_t) nfs_xdr_enc_void,
            // xdr decode function
            (kxdrproc_t) nfs_xdr_dec_void, 0, 0
      },
      { "mnt_mount", // Nombre del procedimiento
            // xdr encode function
            (kxdrproc_t) xdr_encode_dirpath,
            // xdr decode function
            (kxdrproc_t) xdr_decode_fhstatus,
            // Tamaño del buffer de la respuesta
            MNT_dirpath_sz << 2,
            0 // Contador de llamada
      },
      { "mnt_dump", // Nombre del procedimiento
```

```
                // xdr encode function
                (kxdrproc_t) nfs_xdr_enc_void,
                // xdr decode function
                (kxdrproc_t) xdr_decode_mountlist,
                0, // Tamaño del buffer de la respuesta
                0 // Contador de llamada
          },
          { "mnt_umnt", // Nombre del procedimiento
                // xdr encode function
                (kxdrproc_t) xdr_encode_dirpath,
                // xdr decode function
                (kxdrproc_t) nfs_xdr_dec_void,
                0, // Tamaño del buffer de la respuesta
                0 // Contador de llamada
          },
          { "mnt_umntall", // Nombre del procedimiento
                // xdr encode function
                (kxdrproc_t) nfs_xdr_enc_void,
                // xdr decode function
                (kxdrproc_t) nfs_xdr_dec_void,
                0, // Tamaño del buffer de la respuesta
                0 // Contador de llamada
          },
          { "mnt_export", // Nombre del procedimiento
                //xdr encode function
                (kxdrproc_t) nfs_xdr_enc_void,
                //xdr decode function
                (kxdrproc_t) xdr_decode_exports,
                0, // Tamaño del buffer de la respuesta
                0 // Contador de llamada
          },
    };
```

Definición de la versión uno del protocolo mount:

```
    static struct rpc_version  mi_mount_version1 = {
          MI_MOUNT_VERS,        // Número de versión
          MI_MOUNT_PROC,        // Número de procedimientos
          mi_mount_procedures,  // Array de procedimientos
    };
```

Array de versiones del servicio mount:

```
    static struct rpc_version *mi_mount_version[] = {
          NULL,                 // Para la versión cero
          &mi_mount_version1,   // Para la versión uno
    };
```

Estadísticas del protocolo mount:

```
static struct rpc_stat      mi_mount_stats;
```

Declaración del programa mount:

```
struct rpc_program          mi_mount_program = {
   "mount", // Nombre del protocolo
   MI_MOUNT_PROGRAM,   // Número de programa
   // Tamaño
   sizeof(mi_mount_version)/sizeof(mi_mount_version[0]),
   mi_mount_version,   // Array de versiones
   &mi_mount_stats,    // Estadísticas
};
```

5.4.6.1 Null

```
rpc_call(mi_nfs_clnt, nulo, argp, clnt_res, 0);
```

Este procedimiento no realiza ninguna operación. Está disponible para comprobar tiempos de respuesta y el estado del servidor.

5.4.6.2 Montar un directorio

```
struct fhstatus             *clnt_res;

rpc_call(mi_nfs_clnt, montar, path, clnt_res, 0);
```

Este procedimiento sirve para obtener el manejador de un directorio que se encuentra en el servidor. El parámetro del procedimiento es el nombre del directorio del servidor. Este manejador puede usarse en el protocolo de NFS. Este procedimiento también agrega una nueva entrada a la lista de montado del servidor.

5.4.6.3 Obtener la lista de montaje

```
typedef struct mountbody    *mountlist;

struct mountbody {
      name                  ml_hostname;
      dirpath               ml_directory;
      mountlist             ml_next;
};
typedef struct mountbody    mountbody;

void                        *argp;
mountlist                   clnt_res;

rpc_call(mi_nfs_clnt, dump, argp, &clnt_res, 0);
```

Devuelve el listado del sistema de ficheros remoto montado en el servidor, *mountlist* es un puntero a la estructura *mountntbody*, la cual contiene una entrada para cada par máquina – directorio y un puntero al siguiente par.

5.4.6.4 Desmontar un directorio

```
void                            *clnt_res;

rpc_call(mi_nfs_clnt, umnt, path, clnt_res, 0);
```

Borra la entrada dada por el parámetro *path* de la lista de directorios montados.

5.4.6.5 Desmontar todos los directorios montados

```
void                            *argp;
void                            *clnt_res;

rpc_call(mi_nfs_clnt, umntall, argp, clnt_res, 0);
```

Desmonta todos los directorios montados en el cliente.

5.4.6.6 Petición de toda la lista de directorios exportados

```
typedef struct groupnode    *groups;

struct groupnode {
      name                  gr_name;
      groups                gr_next;
};
typedef struct groupnode    groupnode;

typedef struct exportnode   *exports;

struct exportnode {
      dirpath               ex_dir;
      groups                ex_groups;
      exports               ex_next;
};
typedef struct exportnode   exportnode;

void                        *argp;
struct exportnode           exp;

rpc_call(mi_nfs_clnt, exportar, argp, &exp, 0);
```

Devuelve una lista con un número variable de entradas de directorios exportados. Cada entrada contiene un nombre de un directorio exportado del sistema de ficheros y una lista de grupos a los cuales que se le permiten importarlo. El nombre del directorio del sistema de ficheros está en el campo *ex_dir*, y el nombre del grupo está en el campo *ex_groups* de la lista.

6 Desarrollo de módulos del kernel de Linux

El objetivo del capítulo es describir el proceso completo de implementación de un módulo cargable del kernel.

Se realiza un introducción que nos permita diferenciar entre programar un módulo del kernel y una aplicación de usuario para proceder a narrar los componentes necesarios para desarrollar un módulo cargable, continuaremos explicando cómo realizar la asignación dinámica de números mayores, cómo realizar un *makefile* para módulos y cómo podemos exportar símbolos. Se ahondará en la comunicación entre procesos mediante los ficheros especiales de dispositivos definiendo la estructura de operaciones con las distintas operaciones que va a ofrecer el modulo suma.

Se explica el uso de las RPC´s y del servicio portmapper desde el modulo, describiendo los procedimientos del *portmap* y las funciones *XDR* necesarias para finalizar con la implementación del módulo suma donde se especifican las constantes del servicio, los tipos de datos necesarios los procedimientos del servidor y las funciones XDR necesarias para la comunicación por red.

Finalmente se describe la interfaz de usuario implementada en *C* en modo usuario y que nos permitirá hacer invocaciones, *ioctl* a través del dispositivo especial de caracteres, al módulo.

Por lo tanto podemos decir que este capítulo es la base principal que nos permitirá la compresión del siguiente capítulo, así como servirá de base para la implementación del módulo cliente de NFS.

6.1 Introducción

Como ya he comentado, programar un módulo es programar una parte del kernel. Hay diferencias entre programar un módulo del kernel y una aplicación de usuario:

- El **código** de un módulo ha de ser **muy robusto** para evitar caídas del sistema, perdidas de datos, etc. Si por un error de programación pretendemos escribir en una zona de memoria que no es nuestra (por ejemplo, cuando se nos va un puntero de rango) escribiremos donde no queríamos, estropeando lo que allí hubiese y generando un pequeño desastre que hace que se caiga todo el sistema.
- El kernel tiene unos **archivos de cabeceras** para sus librerías que tendremos que incluir (*#include*) en nuestro módulo, tendrán funciones parecidas a las librerías habituales. En un módulo del núcleo sólo puedes usar las funciones del núcleo, no se pueden usar **bibliotecas estándar.**
- Un módulo, **no es una aplicación**, no tiene ninguna función *main()*. En realidad un módulo se parece más a una librería, en él definiremos una serie de funciones, informaremos al kernel de cuales son y qué argumentos necesitan. Cuando el kernel necesite utilizar una función mirará en el módulo y usará la porción de él que necesite. Por eso cuando vayamos a probar nuestro módulo tendremos que hacerlo a través del kernel.
- Para **probar módulos** es muy recomendable generar unos cuantos *scripts* de pruebas que inserten el módulo y ejecuten comandos de forma que el kernel tenga que usar las funciones del módulo. De esta manera podremos ahorrarnos mucho tiempo y además podremos concentrarnos en generar un buen espacio de pruebas para nuestro módulo.

- Al **compilar** el módulo este no se enlaza. Al instalar el módulo este se enlazará con las librerías del núcleo.
- **Deshabilitar las interrupciones.** Puede ser necesario pero hay que habilitarlas posteriormente para no tener que reiniciar el sistema.

6.2 Implementación de un módulo cargable (LKM)

Un módulo del núcleo ha de tener al menos dos funciones: ***init_module()***, que se llama cuando el módulo se inserta en el núcleo, y ***cleanup_module()***, que se llama justo antes de ser quitado. Típicamente, *init_module()* o bien registra un manejador o reemplaza una de las funciones del núcleo con su propio código, normalmente código para hacer algo y luego llamar a la función original. La función *cleanup_module()* deshace lo que *init_module()* ha hecho, debe dar de baja todos los servicios registrados, de forma que el módulo pueda ser descargado de una manera segura.

Ha de incluir los archivos de cabecera necesarios, al menos:

```
#include <linux/kernel.h>
```

Indica que estamos realizando trabajo del núcleo.

```
#include <linux/module.h>
```

Especifica que se trata de un módulo.

A veces tiene sentido dividir el módulo del núcleo en varios ficheros de código, en tal caso:

- Añadir en todos los ficheros fuente menos en uno, *#define __NO_VERSION__*. Normalmente *module.h* incluye la definición de *kernel_version*, una variable global con la versión del núcleo para la que se compila el módulo. Pero *module.h* no lo hará con *__NO_VERSION__*.
- Compilar todos los ficheros fuente.
- Combinar todos los ficheros objeto en uno solo. Bajo *x86*, *ld -m elf_i386 -r -o nombre_del_módulo.o primer_fichero_fuente.o segundo fichero fuente.o.*

Símbolos para el preprocesador, nos permiten acceder a todas las macros definidas en los ficheros de cabeceras del Kernel:

```
#define __KERNEL__
#define MODULE
```

Pueden incluirse en el código aunque es mejor en el *Makefile*.

La función `init_module()` llama a ***register_chrdev()*** para añadir el controlador del dispositivo a la tabla de controladores de dispositivos de carácter del núcleo con un número mayor. Asocia el módulo a un número mayor que no esté reservado para otros dispositivos, la mejor manera de hacer esto es mediante la asignación dinámica de números mayores. Muestra los códigos o comandos de las llamadas *iotls*, devuelve un valor negativo en caso de error y cero en caso de éxito. La cabecera de la función es:

```
int init_module (void)
```

Nota: registra un controlador para el dispositivo especial de caracteres ficticio, no va a estar ligado a ningún hardware.

Además la función *register_chrdev()* da de alta los punteros a función de las operaciones, ***file_operations*** (*&mi_nfs_fops*), que se pueden realizar sobre el dispositivo especial de caracteres (leer, escribir, abrir, cerrar, etc.).

La función *cleanup_module()* se ejecuta al descargar el módulo, desregistra el dispositivo especial de caracteres, ***unregister_chrdev()*** da de baja nuestro módulo en la lista de dispositivos de carácter. Una vez ejecutada esta función, cualquier acceso a */dev/mi_nfs* nos dará un error, pues ningún módulo resuelve las *file_operations* de ese fichero.

Si por alguna razón *unregister_chrdev()* no es capaz de dar de baja el módulo, porque le pasemos un mayor diferente por ejemplo, retornará un –*EINVAL*. La cabecera de la función es:

```
void cleanup_module (void)
```

Cleanup_module es una función void, una vez que se llama el módulo está muerto. Hay un contador que cuenta cuántos otros módulos del núcleo están usando el módulo, llamado contador de referencia, que es el último número de la línea en */proc/modules*. Si este número es distinto de cero, *rmmod* fallará. La cuenta de referencia del módulo está disponible en la variable *mod_use_count_*. Hay macros definidas para manejar esta variable, *MOD_INC_USE_COUNT* y *MOD_DEC_USE_COUNT*.

Para poder utilizar estas dos funciones es necesaria la inclusión del fichero de cabeceras *linux/fs.h*. Registrar y liberar es la funcionalidad general de estas dos funciones. Los módulos son llamados por procesos a través de las llamadas al sistema, por los dispositivos hardware a través de las interrupciones, o por otras partes del núcleo, simplemente llamando a funciones específicas. Cuando se añade código al núcleo, es para registrarlo como parte de un manejador o para un cierto tipo de evento y lo liberamos al quitarlo.

6.3 Asignación dinámica de números mayores

A la hora de insertar un módulo podemos hacer que este se asocie a un número mayor que no esté en uso. El problema es que no sabemos qué número mayor ha escogido hasta que el módulo no ha sido insertado, por lo que sí queremos crear dispositivos asociados a este número mayor, tendremos que esperar a insertar primero el módulo, averiguar qué número mayor se le ha otorgado y luego crear los dispositivos asociado a este número mayor.

Si no queremos hacer esto, lo único que podemos hacer es buscar un número mayor que no esté reservado para ningún dispositivo y utilizar este para nuestro módulo. En *Documentation/devices.txt* hay una lista completa de los números mayores que están asociados a los servicios más comunes.

He usado la asignación dinámica de números mayores, para ello cuando el módulo se da de alta, *init_module()*, tenemos que decirle que pruebe diferentes números mayores hasta que encuentre uno que esté sin usar.

Una vez insertado el módulo podemos ver qué número mayor se le ha asignado, usando el sistema de ficheros `/proc`:

```
$ cat /proc/devices

Character devices:
254   suma
```

Ahora ya sé que número mayor se nos ha asignado, por lo que puedo crear el dispositivo, con su número mayor y menor, con toda confianza:

```
$ mknod /dev/suma c 254 0
```

Puedo crear varios dispositivos, cada uno con un número menor diferente.

```
$ mknod /dev/suma c 254 1
```

Evito tener que cargar el módulo para consultar el número mayor asignado y poder crear el dispositivo, generando un *script* para cargar el módulo que obtenga el número mayor asignado y cree el dispositivo especial.

6.4 Makefile para los módulos del núcleo

Creamos un Makefile para estandarizar la compilación de nuestro código.

Un módulo del núcleo no es un ejecutable independiente, sino un fichero objeto que será enlazado dentro del núcleo en tiempo de ejecución. En consecuencia, deberían ser compilados con la bandera *-c*. Todos los módulos del núcleo deben ser compilados con ciertos símbolos definidos.

- ***__KERNEL__***: Indica a los ficheros de cabeceras que este código se ejecutará en modo kernel, y no como parte de un proceso de usuario, modo usuario.
- ***MODULE***: Pide a los ficheros de cabeceras que le den las definiciones apropiadas para un módulo del núcleo.
- ***LINUX***: Técnicamente hablando, esto no es necesario. Solo para escribir un módulo que se compile en más de un sistema operativo. Esto te permitirá hacer compilación condicional en las partes que son dependientes del sistema operativo.

Hay otros símbolos que deben ser incluidos, dependiendo de cómo se haya compilado el núcleo, esto se puede ver en `/usr/include/linux/config.h`.

- ***__SMP__***: Multiproceso simétrico. Debe estar definido si el núcleo fue compilado para soportar multiproceso simétrico, incluso sí sólo se está ejecutando en una CPU.
- ***CONFIG_MODVERSIONS***: Si está habilitado, necesita estar definido cuando se compile el módulo e incluir */usr/include/linux/modversions.h.* Esto también puede ser realizado por el propio código.

Hay que indicar al compilador donde están las cabeceras del kernel: *-I/usr/src/linux/include*. Entre los sitios por defecto donde el compilador, *gcc*, busca librerías no está el directorio de las

cabeceras del kernel, por lo que no será capaz de encontrar los ficheros incluidos, *module.h* y *kernel.h*.

Vamos a incluir las definiciones de *__KERNEL__* y *MODULE* en el comando de compilación, así nos las incluirá el compilador. Es recomendable quitar su definición de nuestro fichero fuente, no pasa nada por que estén en los dos lados, pero el compilador lanza dos advertencias por redefinición de símbolos, con lo que puede contribuir a crear confusión sobre el resultado de la compilación

Un error de programación en un módulo puede ser fatal para el sistema, por eso vamos a decirle al compilador que sea lo más estricto posible, -*Wall*, es decir que nos muestre los *"WARNING"*. Esto nos ayudará a construir un código libre de errores. Hemos de depurar nuestro código **hasta que el compilador no muestre ninguna advertencia**.

Es muy habitual en el kernel declarar funciones del tipo ***inline***. Por defecto *gcc* no expande las funciones así definidas si no le pides que haga optimización de tu código, por eso conviene utilizar la opción -*O* para el *gcc*. Para aquellos que acostumbren a confiar en el compilador para optimizar su código al máximo usando -*O2*, han de saber que usar esta opción con módulos es bastante arriesgado.

Siguiendo estas indicaciones podemos generar un **Makefile** con el siguiente código:

```
CC = gcc
MODCFLAGS := -Wall -DMODULE -D__KERNEL__ -DLINUX

suma.o:    suma.c /usr/include/linux/version.h
           $(CC) $(MODCFLAGS) -c suma.c
           echo insmod suma.o //para insertarlo
           echo rmmod suma //para descargarlo
```

En mi caso he optado por crear tres *scripts*, uno para compilar el módulo, otro para cargarlo en el núcleo y otro para descargarlo:

El comando para compilar exitosamente el módulo:

```
gcc    -D__KERNEL__    -DMODULE    -O    -Wall    -
I/usr/src/linux/include -c suma.c
```

Con esto generamos el fichero objeto *suma.o*, que es directamente el módulo a insertar.

Cargar el módulo:

```
#!/bin/bash
#carga_modulo
modulo = suma
dispositivo = /dev/suma
permisos = 666

#Antes de cargar el módulo, sincronizo
sync
```

```
#inserto el modulo y consulto el número mayor
/sbin/insmod ${modulo}.o
mayor = `cat /proc/devices |grep ${modulo} | cut -d' ' -
f1`

#Borro el dispositivo de anteriores pruebas y creo el
nuevo con su número mayor obtenido
rm -f ${dispositivo}

#creo un dispositivo de carácter porque me da igual que
sea de carácter que de bloque
mknod ${dispositivo} c ${mayor} 0
chmod ${permisos} ${dispositivo}
```

Descargar el módulo:

```
#!/bin/bash
#descarga_modulo
modulo = suma

#Antes de descargar el módulo, sincronizo
sync

#descargo el modulo
/sbin/rmmod ${modulo}
```

Cuando generamos el código objeto del módulo, todas las referencias a símbolos del kernel, por ejemplo *printk*, están sin resolver. Es en el momento de la inserción del módulo cuando se resuelven dichas referencias, ya que al cargar un módulo en el kernel estamos linkando la imagen en memoria del kernel al código objeto del módulo.

Para que se puedan resolver esas referencias a símbolos del kernel, este publica todas sus referencias a través del sistema de ficheros virtual *proc*. Para ver la tabla de símbolos del kernel ejecutar la siguiente orden:

```
$ cat /proc/ksyms
```

Cuando insertamos un módulo, éste ofrece como símbolos por defecto todos sus símbolos globales no estáticos, estos nuevos símbolos pueden ser usados por otros módulos, por eso la salida de este comando incluye no solo símbolos del kernel si no los de aquellos módulos que estén cargados en el momento de ejecución del comando.

Para buscar un determinado símbolo ejecutar: *cat /proc/ksyms | grep símbolo*, para ver todos los símbolos que exporta nuestro módulo, una vez insertado hacemos:

```
$ cat /proc/ksyms | grep suma
```

Nota: El módulo del núcleo será borrado cuando *root*, el superusuario, quiera. El motivo es que si el fichero del dispositivo es abierto por un proceso y entonces quitamos el módulo del

núcleo, el uso del fichero causaría una llamada a la posición de memoria donde la función apropiada usada debería estar. Si tenemos suerte, ningún otro código fue cargado allí, y obtendremos un feo mensaje. Si no tenemos suerte, otro módulo del núcleo fue cargado en la misma posición, lo que significará un salto en mitad de otra función del núcleo. El resultado sería imposible de predecir, pero no sería positivo.

6.5 Exportar símbolos

Nuestro módulo puede exportar una serie de símbolos que pueden ser usados por otros módulos. Por defecto, todas aquellas variables definidas como globales no estáticas serán exportadas.

Pero en general no queremos exportar todas nuestras variables globales, y definir como estáticas las que no queremos exportar no es una solución aceptable. Por ello el kernel dispone de un mecanismo para dar de alta los símbolos que nosotros elijamos.

Al eliminar el módulo de la memoria estos símbolos exportados desaparecen, por lo que no es necesario darlos de baja desde *cleanup_module()*.

La interfaz que ofrece el kernel para dar de alta símbolos es el siguiente:

- Definimos todas nuestras variables globales, tanto estáticas como no estáticas (solo podremos exportar las globales no estáticas).
- Si queremos exportar todas las variables no estáticas, no habrá problema, este es el comportamiento por defecto.
- Si queremos exportar solo algunas de las variables no estáticas, definimos ***EXPORT_SYMTAB*** y mediante la macro ***EXPORT_SYMBOL(símbolo)*** vamos exportando todas las variables no estáticas que queramos. El resto no serán exportadas.
- Si no queremos exportar ninguna, usar la macro ***EXPORT_NO_SYMBOLS***.

El uso de estas macros viene detallado en */linux/module.h*.

6.6 Multiproceso simétrico

Una de las formas más fáciles y baratas de aumentar el rendimiento del hardware es poner más de una CPU en la placa. Esto se puede realizar haciendo que *CPUs* diferentes tengan trabajos diferentes, multiproceso asimétrico, o haciendo que todos se ejecuten en paralelo, realizando el mismo trabajo, multiproceso simétrico o SMP. El hacer multiproceso asimétrico requiere un conocimiento especializado sobre las tareas que la computadora debe ejecutar. En cambio el multiproceso simétrico es relativamente fácil de implementar.

En un entorno de multiproceso simétrico, las CPUs comparten la misma memoria, y como resultado, el código que corre en una CPU puede afectar a la memoria usada por otra. No se puede asegurar que una variable que ha sido establecida a un cierto valor en la línea anterior todavía tenga el mismo valor, la otra CPU quizás la haya modificado. Obviamente, es imposible programar algo de esta manera.

En el caso de la programación de procesos esto no suele ser un problema, porque un proceso normalmente sólo se ejecutará en una CPU a la vez. El núcleo, sin embargo, podría ser llamado por diferentes procesos ejecutándose en CPUs diferentes.

En la versión 2.0.x, esto no es un problema porque el núcleo entero está en un gran "*spinlock*". Esto significa que si una CPU está dentro del núcleo y otra CPU quiere entrar en él, por ejemplo por una llamada al sistema, tiene que esperar hasta que la primera CPU haya acabado. Esto es lo que hace al SMP en Linux seguro, pero terriblemente ineficiente.

En la versión 2.2.x, varias CPUs pueden estar dentro del núcleo al mismo tiempo, algo que los programadores de módulos tienen que tener en cuenta.

6.7 Comunicación con los procesos

Hay dos formas principales de que un módulo del núcleo se comunique con los procesos. Una es a través de los ficheros especiales de dispositivos, como los que están en el directorio */dev*, y la otra es usar el sistema de ficheros */proc*.

6.7.1 Ficheros especiales de dispositivos

Uno de los principales motivos por el que programar en el núcleo es dar soporte a algún tipo de hardware. El propósito original de los ficheros de dispositivo es permitir a los procesos comunicarse con los controladores de dispositivos en el núcleo, y a través de ellos con los dispositivos físicos: módems, terminales, etc.

A cada controlador de dispositivo, que es responsable de algún tipo de hardware, se le asigna su propio número mayor. La lista de los controladores y de sus números mayores está disponible en */proc/devices*. A cada dispositivo físico administrado por un controlador de dispositivo se le asigna un número menor. El directorio */dev* contiene un fichero especial, llamado fichero de dispositivo, para cada uno de estos dispositivos, tanto si está realmente instalado en el sistema como sino.

No existe un motivo técnico por el que tienen que estar en el directorio */dev*, es sólo una convención útil.

El controlador del dispositivo se compone de una serie de funciones *device_acción*, que se invocan cuando se actúa sobre un fichero de dispositivo asociado al número mayor. La forma en que el núcleo sabe cómo llamarlas es a través de la estructura *file_operations*, que se da cuando el dispositivo es registrado, e incluye punteros a esas funciones.

La estructura *file_operations* está definida en *include/linux/fs.h*.

6.7.2 File_operations

A continuación se muestra la declaración de la estructura para el ejemplo de la suma:

```
struct file_operations suma_fops =
{
      NULL,                          //lseek
      NULL,                          //read
      NULL,                          //write
      NULL,                          //readdir
      NULL,                          //poll
      ioctl: (void (*)) suma_ioctl,  //ioctl
      NULL,                          //mmap
      open: suma_open,               //open
      NULL,                          //flush
      release: suma_release,         //realease
      NULL,                          //fsync
      NULL,                          //fasync
      NULL,                          //check_media_change
      NULL,                          //revalidate
      NULL,                          //lock
};
```

- ***lseek:*** Se usa para cambiar la posición actual de lectura-escritura sobre el fichero. La nueva posición se retorna en forma de un entero positivo, si el valor retornado es negativo, es que se ha producido algún error. El objeto de esta función es por tanto modificar la estructura *file* del fichero del dispositivo para que refleje la nueva posición de lectura-escritura.
- ***read****:* Se usa para extraer datos desde el dispositivo. Si se apunta a *null*, cualquier lectura sobre el dispositivo dará un *-EINVAL*. Un resultado positivo informa de la cantidad de datos leídos.
- ***write****:* Manda datos al dispositivo. Se hacen las mismas consideraciones que para *read*.
- ***readdir:*** Debe apuntar a *null*, solo se usa para directorios, no para dispositivos.
- ***poll****:* Para dejar en espera al proceso llamante.
- ***ioctl****:* Es la función **comodín**, implementa todas aquellas cosas para las que no hay una función específica. Estas funcionalidades adicionales pueden depender del dispositivo, y pueden ser muchas, los argumentos numéricos a **ioctl* sirven para elegir entre las posibles opciones y para el paso de argumentos.
- ***mmap****: mapea* la memoria del dispositivo a memoria del proceso llamante, útil para DMA, si no se implementa retorna *–EINVAL*.
- ***open****:* Es la primera operación que se realiza sobre el fichero del dispositivo, será la encargada de las inicializaciones.
- ***release****:* Esta operación se invoca cuando el fichero deja de utilizarse. Algo así como el contrario de *open*.
- ***check_media_change****:* Solo se usa con dispositivos de bloque, especialmente para dispositivos con medio extraíble, como la disquetera. Se usa para ver si se cambió el medio desde la última operación.
- ***revalidate****:* Solo para dispositivos de bloque, está relacionada con el manejo de la caché de buffers.
- ***lock****:* Para bloquear accesos.

6.7.2.1 ioctl

Los ficheros de dispositivos se supone que representan dispositivos físicos. La mayoría de los dispositivos físicos se utilizan para salida y para entrada.

Todas aquellas funciones que no están definidas en *file_operations* se implementan con la función especial IOCTL, *input output control*.

```
int  funcion_ioctl(struct  inode  *pinodo,  struct  file
*pfile, unsigned int comando, unsigned long argumentos)
```

Los argumentos de entrada a *ioctl()* son el *inodo* del dispositivo, la estructura *file* del dispositivo asociada al proceso que la invocó, un argumento numérico que identificará cuál de todas las posibles acciones se quiere realizar y por último otro argumento numérico, en realidad un puntero, en el que irán los posibles argumentos de las diferentes acciones a realizar, en el caso de que fuesen necesarios.

Cada dispositivo tiene sus propias órdenes *ioctl*, que pueden leer *ioctl's* (para enviar información desde un proceso al núcleo), escribir *ioctl's* (para devolver información a un proceso), ambas o ninguna. La función se invoca con tres parámetros: el descriptor de fichero del dispositivo, el número de la *ioctl*, y un parámetro (si se quiere pasar), que es de tipo *long*. Para pasar otros tipos de datos se puede hacer una conversión, *cast*.

Ejemplos de llamadas ioctl:

```
resultado = ioctl(file, 0x101);
resultado = ioctl(file, 0x101, &param);
```

El identificador numérico del comando se construye concatenando cuatro campos de bits en un *unsigned int*:

- ***type***: El número mágico del dispositivo, clave para comprobar que el proceso que llama a *ioctl* conoce realmente cual es el *comando* para cada acción asociada al dispositivo. Se escoge uno al azar.
- ***number***: El número que identifica a cada acción.
- ***direcction***: Para las acciones que realicen transferencia de datos, este campo nos indica el sentido de la transferencia desde el punto de vista de la aplicación. Este campo puede tener uno de los siguientes valores:

    - ***_IOC_NONE***: No habrá transferencia de datos.
    - ***_IOC_READ***: Se leerán datos del dispositivo.
    - ***_IOC_WRITE***: Se mandarán datos al dispositivo.

- ***size***: El tamaño de los datos transferidos, si procede, en unidades del valor de la macro ***_IOC_SIZEBITS***.

Para la construcción de cada *comando*, la librería ***asm/ioctl.h*** nos proporciona las siguientes macros:

- ***_IO(type, nr)***: para cuando no hay transferencia de datos.
- ***_IOR(type ,nr, size)***: para cuando se quieren obtener datos.
- ***_IOW(type, nr, size)***: para cuando se quieren mandar datos.
- ***_IOWR(type, nr, size)***: para cuando se quieren mandar y obtener datos.

También se definen unas macros para decodificar el comando:

- ***_IOC_TYPE(nr)***: Nos da el *type* a partir del *comando*.
- ***_IOC_NR(nr)***: Nos da el *number* a partir del *comando*.
- ***_IOC_DIR(nr)***: Nos da el *direcction* a partir del *comando*.
- ***_IOC_SIZE(nr)***: Nos da el *size* a partir del *comando*.

Para usar *ioctls* en los módulos del núcleo, es mejor recibir una asignación *ioctl* oficial, por si accidentalmente coges los *ioctls* de alguien, o alguien coge los tuyos. Para más información, consulta el árbol del código fuente del núcleo en *Documentation/ioctl-number.txt*.

Cabecera de la función ioctl para la suma:

```
int suma_ioctl(struct inode *pinodo, struct file *pfile,
unsigned int comando, struct parametros *argumentos)
```

Donde:

```
struct mis_param {
	int			a;
	int			b;
};

struct parametros {
	struct sockaddr_in	dir_ser;
	// Máquina del servidor
	char			*maquina;
	struct mis_param	param;
};
```

Función del módulo que recibe las llamadas *ioctls* con sus correspondientes parámetros. Realiza las comprobaciones pertinentes del comando recibido, tales como que el *type* y el *number*, número que identifica cada acción, sean correctos. Por medio del comando recibido en la llamada ioctl se identifica la acción a realizar, en este caso la única posible es la suma. Obtiene el puerto por medio del *portmapper* a través del número de programa, la versión, el protocolo de transporte y la dirección de la máquina servidor. Pone en formato de red los valores necesarios tales como el puerto obtenido y los valores a sumar e invoca la función encarga de dicha acción. Por último controla el estado de la ejecución en el servidor para informar del mismo al término de la ejecución.

6.7.2.2 open

Se encarga de abrir el fichero del dispositivo para poder interactuar con él. Por medio del fichero especial de caracteres hacemos llegar las ioctls al módulo.

La cabecera de la función open de la suma quedaría de la siguiente manera:

```
int suma_open (struct inode *inode, struct file *file)
```

Se encarga de comprobar que el número menor del dispositivo es correcto y de incrementar el contador de uso, *MOD_INC_USE_COUNT.*

MOD_INC_USE_COUNT es una macro definida en *include/linux/module.h*, que ayuda al programador de módulos a manejar de forma sencilla la cuenta de uso del módulo. Su uso en *open* es fundamental si se quiere evitar que nos desmonten el módulo mientras está siendo usado.

Se invoca de la siguiente manera:

```
file = open("/dev/suma", O_RDWR);
```

6.7.2.3 release

Se invoca cuando el fichero deja de utilizarse.

La cabecera de la función release de la suma quedaría de la siguiente manera:

```
int suma_release (struct inode *inode,struct file *file)
```

Comprueba que el número menor del dispositivo es correcto y decrementa el contador de uso, *MOD_DEC_USE_COUNT.*

Release decrementa la cuenta de uso, **MOD_DEC_USE_COUNT,** si queremos que una vez que el módulo no esté siendo utilizado se pueda eliminar de la memoria satisfactoriamente.

Se invoca de la siguiente manera:

```
result = close(file);
```

6.7.3 Número mayor y menor del dispositivo

Un determinado driver puede atender a varios dispositivos, todos los que tengan el mismo número mayor que el driver. Podemos hacer código condicional al dispositivo dentro del módulo conociendo el número menor del dispositivo que provocó la llamada.

El número menor está almacenado en el **inodo** del fichero. Cuando se invoca una operación sobre ese fichero, al módulo correspondiente se le pasa un puntero al inodo, siendo así posible la identificación del fichero que provocó la llamada.

El campo **i_dev** del inodo contiene la información sobre el número menor y el número mayor de dicho inodo. El tipo de datos de *i_dev* es **kdev_t**. Para poder extraer de forma sencilla estos datos, en *linux/kdev_t.h* se definen macros que nos pueden ser de utilidad:

- ***MAJOR(kdev_t dev)***: Extrae de un tipo de datos *kdev_t* el número mayor (unsigned int).
- ***MINOR(kdev_t dev)***: Extrae de un tipo de datos *kdev_t* el número menor (unsigned int).
- ***MKDEV(int ma, int mi)***: Genera una variable de tipo *kdev_t* a partir del número mayor y el menor.

Gracias a esto podemos crear diferentes "dispositivos lógicos" que sean en realidad diferentes formas de acceder a un mismo "dispositivo físico".

6.7.4 El sistema de ficheros /proc

Existe otro mecanismo para que el núcleo y los módulos del núcleo envíen información a los procesos, el sistema de ficheros */proc*. Originalmente diseñado para permitir un fácil acceso a la información sobre los procesos, ahora lo utiliza cualquier elemento del núcleo que tiene algo interesante que informar, como */proc/modules* que tiene la lista de los módulos y */proc/meminfo* que tiene las estadísticas de uso de la memoria.

El método para usar el sistema de ficheros */proc* es muy similar al usado con los controladores de dispositivos: se crea una estructura con toda la información que necesita el fichero */proc*, incluyendo punteros a cualquier función manejadora. *Init_module()* registra la estructura con el núcleo y *cleanup_module()* la libera.

El sistema de ficheros */proc* se escribió principalmente para permitir al núcleo informar de su situación a los procesos. En Linux hay un mecanismo estándar para el registro de sistemas de ficheros. Como cada sistema de ficheros tiene que tener sus propias funciones para manejar las operaciones de inodos, tratan con las formas de referenciar el fichero, y ficheros, tratan con el fichero, la estructura *proc_dir_entry*, en *proc_fs.h*, contiene los punteros a *struct inode_operations*, y a *struct file_operations*.

6.8 Uso de RPC desde un módulo

En este apartado se describe como se declaran y utilizan las RPCs en el kernel.

En el kernel no hay ningún un precompilador de RPC, como *rpcgen*, que nos genera automáticamente el código fijo y relacionado con la parte de comunicaciones, es decir que nos genere los *stubs* o suplentes. El programador debe definir los distintos programas o servicios que va a utilizar. Por tanto debe generar un conjunto de llamadas de RPC, *Remote Procedure Call*, y XDR, *eXternal Data Representation*. Esto requiere mucho conocimiento, hay muchas causas de errores, mucha inversión en tiempo dedicado a problemas de comunicaciones a parte del problema en si a resolver.

Para permitir que un módulo del kernel realice llamadas RPC a un servidor ha de implementar el servicio *portmap*, el cual permite registrar servicios en el servidor asociándolos a un puerto por el que escuchará peticiones, desregistra estos servicios así como permite su consulta.

6.8.1 Protocolo del programa PortMapper

Es necesario el uso del programa *portmapper* para obtener los puertos por los que escucha peticiones el servidor.

Definición del servicio *portmap* en el módulo del núcleo:

Define los procedimientos implementados por el servidor *portmapper*, así como las funciones de codificación y decodificación de los parámetros utilizados por cada uno de dichos procedimientos:

```
static struct rpc_procinfo pmap_procedures[4] = {
      { "pmap_null",
            (kxdrproc_t) xdr_error,
            (kxdrproc_t) xdr_error, 0, 0 },
      { "pmap_set",
            (kxdrproc_t) xdr_encode_mapping,
            (kxdrproc_t) xdr_decode_bool, 4, 1 },
      { "pmap_unset",
            (kxdrproc_t) xdr_encode_mapping,
            (kxdrproc_t) xdr_decode_bool, 4, 1 },
      { "pmap_get",
            (kxdrproc_t) xdr_encode_mapping,
            (kxdrproc_t) xdr_decode_port, 4, 1 },
};
```

Definición de la versión dos:

```
static struct rpc_version  pmap_version2 = {
      2,  4,  pmap_procedures  //  Versión,  número  de
      procedimientos, procedimientos de la versión
};
```

Array de versiones:

```
static struct rpc_version *pmap_version[] = {
      NULL,
      NULL,
      &pmap_version2,
};
```

Estadísticas del portmap:

```
static struct rpc_stat           pmap_stats;
```

Declaración del programa:

```
struct rpc_program    pmap_program = {
      "portmap",                 // Nombre del protocolo
      RPC_PMAP_PROGRAM,          // Número de programa
      // Tamaño
      sizeof(pmap_version)/sizeof(pmap_version[0]),
      pmap_version,              // Array de versiones
      &pmap_stats,               // Estadísticas
};
```

6.8.1.1 pmap_null

```
rpc_call(suma_clnt, pmap_nulo, argp, clnt_res, 0);
```

Este procedimiento no realiza ninguna operación. Por convenio, el procedimiento cero de cualquier programa no toma ningún parámetro, ni devuelve ningún resultado.

6.8.1.2 pmap_set

```
rpc_call(suma_clnt, PMAP_SET, &param, res, 0);
```

Cuando un programa se quiere utilizar en una máquina, lo primero que tiene que hacer es registrarse en el *portmapper* de la misma máquina. Esta función pasa como parámetro de entrada la estructura *rpc_portmap*, definida en *include/linux/sunrpc/clnt.h*, que contiene el número de programa, el número de versión, el protocolo de transporte y el puerto donde espera peticiones de servicio. Devuelve la respuesta en función del éxito de la conexión.

6.8.1.3 pmap_unset

```
rpc_call(suma_clnt, PMAP_UNSET, &param, res, 0);
```

Cuando un programa se hace inaccesible, debe borrar su registro del *portmapper* de la máquina. Los parámetros y resultados tienen significados idénticos a los de *pmap_set*. Se ignoran los campos del protocolo de transporte y del número de puerto.

6.8.1.4 pmap_get

```
rpc_call(pmap_clnt, PMAP_GETPORT, &map, &map.pm_port,0);
```

Dado un número de programa, número de la versión y el número del protocolo de transporte, esta llamada devuelve el número de puerto en el que el programa espera peticiones de servicio. Un valor de puerto cero indica que el programa no ha sido registrado. El campo del puerto se ignora de los argumentos.

6.8.1.5 Funciones XDR

Cuando programamos una aplicación distribuida debemos tener en cuenta que puede correr en máquinas diferentes, que probablemente codifican los números de forma diferente. Así por ejemplo, un número entero se codifica con los bytes más significativos en primer lugar en una máquina con CPU *Intel,* mientras que una máquina con CPU *Motorola* pone los bytes más significativos al final. Cuando recibimos o transmitimos datos, entre ellos las direcciones IP y puertos, debemos utilizar un lenguaje común. Para ello disponemos de una serie de funciones que convierten nuestro sistema de codificación a uno estándar que llamaremos **codificación de red**. El nombre de cada función específica el tipo de conversión realizada, así la función ***htonl*** convierte un entero largo, *long*, (dirección IP) de nuestra máquina, *host*, a la codificación de red, *net*. Debemos cuidar de utilizar siempre estas funciones, de esta manera nuestra aplicación podrá recompilarse para otras máquinas sin problemas.

Estas funciones convierten datos en formato hosts a formato de red y viceversa:

```
in_addr_t htonl(in_addr_t hostlong);
in_port_t htons(in_port_t hostshort);
in_addr_t ntohl(in_addr_t netlong);
in_port_t ntohs(in_port_t netshort);
```

A continuación se muestran las cabeceras de las funciones de codificación y decodificación utilizadas por el portmapper y se explica su cometido.

6.8.1.5.1 pmap_null

```
static int xdr_error(struct rpc_rqst *req, u32 *p, void
*dummy)
```

Esta función simplemente devuelve un código de error.

6.8.1.5.2 pmap_set y pmap_unset

```
static int xdr_encode_mapping(struct rpc_rqst *req, u32
*p, struct rpc_portmap *map)
```

Cambia a formato de codificación de red los valores de *rpc_portmap*, es decir el número de programa, la versión, el protocolo de transporte y el puerto.

```
static int xdr_decode_bool(struct rpc_rqst *req, u32 *p,
unsigned int *boolp)
```

Cambia a formato de codificación de máquina el valor devuelto por el servidor.

6.8.1.5.3 pmap_get

```
static int xdr_encode_mapping(struct rpc_rqst *req, u32
*p, struct rpc_portmap *map)
```

Explicado en el apartado anterior.

```
static int xdr_decode_port(struct rpc_rqst *req, u32 *p,
unsigned short *portp)
```

Cambia a formato de codificación de máquina el valor devuelto por el servidor, el número de puerto solicitado.

6.9 El programa calcular

En este apartado se describe y muestra la implementación del servicio suma.

6.9.1 Constantes del servicio

```
#define SUMA_PROGRAM  100022254 // Número de programa
#define SUMA_PORT     643 //Puerto para UDP
#define SUMA_VERS     1 // Número de versión
#define SUMA_PROC     2 // Número de procedimientos
#define sumar         1 //Número del procedimiento sumar
```

6.9.2 Tipos de datos

Estructura para enviar la petición al servidor y recoger la respuesta de este:

```
struct rpc_param {
      __u32       a;
      __u32       b;
      __u32       result;
};
```

Donde "a" y "b" son los números con los que vamos a operar y *result* el resultado obtenido.

6.9.3 Procedimientos del servidor

A continuación se definen los procedimientos remotos proporcionados por el servicio:

Declaración del servicio suma:

```
static struct rpc_procinfo  suma_procedures[2] = {
      {     "nulo",
            //xdr encode function
            (kxdrproc_t) xdr_error,
            //xdr decode function
            (kxdrproc_t) xdr_error, 0, 0
      },
      {     "calcular_1",// Nombre del procedimiento
            //xdr encode function
            (kxdrproc_t) xdr_encode_param,
            //xdr decode function
            (kxdrproc_t)xdr_decode_result, 2, 1
      },
};
```

Definición de la versión uno del servicio:

```
static struct rpc_version   suma_version1 = {
      SUMA_VERS, // Número de versión
      SUMA_PROC, // Número de procedimientos
      suma_procedures, // Array de procedimientos
};
```

Array de versiones del servicio:

```
static struct rpc_version *suma_version[] = {
      NULL, // Para la versión cero
      &suma_version1, // Para la versión 1
};
```

Estadísticas del servicio:

```
static struct rpc_stat      suma_stats;
```

Declaración del programa:

```
struct rpc_program    suma_program = {
      "calcular", // Nombre del programa
      SUMA_PROGRAM, // Número de programa
      // Tamaño
      sizeof(suma_version)/sizeof(suma_version[0]),
      suma_version, // Array de versiones
      &suma_stats, // Estadísticas
};
```

6.9.3.1 nulo

```
rpc_call(suma_clnt, nulo, argp, clnt_res, 0);
```

Este procedimiento no realiza ninguna operación. Está disponible para comprobar tiempos de respuesta y el estado del servidor.

6.9.3.2 calcular_1

```
rpc_call(suma_clnt, sumar, &param, &param.result, 0);
```

Este procedimiento invoca la petición al servidor para que realice la suma de los dos números pasados como parámetro. Pone en formato de red los parámetros y recoge las respuestas del servidor convirtiendo estas a formato máquina.

6.9.4 Funciones XDR

A continuación se muestran las cabeceras de las funciones de codificación y decodificación utilizadas por la suma y se explica su cometido.

6.9.4.1 nulo

```
static int xdr_error(struct rpc_rqst *req, u32 *p, void
*dummy)
```

Ya está explicado en el apartado funciones xdr del portmapper.

6.9.4.2 calcular_1

```
static int xdr_encode_param (struct rpc_rqst *req, u32
*p,struct mis_param *objp)
```

Cambia a formato de codificación de red los valores de los números pasados a la petición.

```
static int xdr_decode_result(struct rpc_rqst *req, u32
*p, int *result)
```

Cambia a formato de codificación de maquina el resultado de obtenido de la petición suma al servidor.

6.10 Interfaz de usuario

La interfaz de usuario esta implementada en un programa en modo usuario que permite realizar las llamadas *ioctl* a través del dispositivo especial de caracteres para hacer llegar las peticiones al módulo del núcleo. Una vez cargado el módulo en el núcleo se pueden invocar dichas llamadas.

6.10.1 Proceso en modo usuario

Gracias a este proceso podremos configurar las operaciones que ha de realizar el núcleo sin tener que modificarlo. Por medio del proceso podremos invocar la función *ioctl,* como ya he comentado, se invoca con tres parámetros: el descriptor de fichero del dispositivo, el número de la *ioctl*, y un parámetro (si se quiere pasar), que es de tipo long. Para pasar otros tipos de datos se puede hacer una conversión, *cast*.

Para ello dicho proceso ha de abrir el fichero especial de dispositivo o dispositivo especial de caracteres, para obtener el descriptor de fichero necesario para invocar la *ioctl.* Una vez que se haya cargado el módulo podremos ver y utilizar los números de las distintas *ioctls* necesarios para el segundo parámetro. Además de introducir los parámetros necesarios en la estructura para realizar la petición al servidor.

La interfaz de usuario para el módulo **suma** vendrá definida por un programa que ejecuta en modo usuario con una función principal:

```
int main()
```

Que abre el dispositivo especial de caracteres por medio de la llamada al sistema:

```
file = open("/dev/suma", O_RDWR);
```

Obtiene la dirección de la máquina servidor, introduce los valores necesarios en la estructura declarada para realizar las llamadas *ioctls*, tales como la máquina del servidor y los valores de la suma a realizar. Invoca la llamada ioctl:

```
result = ioctl(file, 0x101, &param);
```

Esta llamada se encarga de transferir la ejecución al módulo del kernel a través del dispositivo especial de caracteres por medio de la estructura *file_operations* asociada. Devuelve el resultado obtenido por el servidor.

Controla el resultado de la ejecución de la llamada *ioctl* y por tanto de la comunicación con el módulo y de este con el servidor. Cierra el dispositivo especial de caracteres por medio de la llamada al sistema:

```
result = close(file);
```

Por último retornando el estado de la ejecución, tanto en caso de error como de éxito.

7 Diseño e implementación del módulo cliente de NFS

A continuación se describe el diseño y la implementación del módulo de Linux para el acceso a servidores NFS versión 2.

En el capítulo anterior se ha descrito la implementación de un módulo de Linux que actúa como cliente del programa calcular. La mayoría de los apartados anteriormente comentados se pueden aplicar, con mínimas modificaciones, al módulo cliente de NFS, véase la implementación de módulos cargables (*init_module* y *cleanup_module*), el programa *portmapper*, la asignación dinámica de números mayores, el *makefile*, la estructura *file_operations*, etc. En el capítulo quinto se han descrito los servicios de los protocolos NFS y MOUNT, véase las constantes del servicio, tamaño de las estructuras XDR, tipos de datos básicos, procedimientos y funciones. Por este motivo este capítulo se centra en la interfaz de usuario implementada para los protocolos MOUNT y NFS, el acceso al módulo a través del dispositivo especial de caracteres, la implementación de las llamadas ioctl y de las operaciones del portmapper en los protocolos MOUNT y NFS así como las funciones y los tipos de datos XDR desarrollados para ambos protocolos.

7.1 Interfaz de usuario

La interfaz de usuario para el módulo **mi_nfs** vendrá definida por un programa en modo usuario con una función:

```
int main()
```

Que abre el dispositivo especial de caracteres por medio de la llamada al sistema:

```
file = open("/dev/mi_nfs", O_RDWR);
```

Inicializa las estructuras, reserva memoria para los *filehandle* e introduce los valores necesarios en la estructura parámetros que se pasa como argumento en las llamadas ioctls, tales como la máquina del servidor y los datos necesarios para las distintas operaciones. Invoca las llamadas ioctl y controla el resultado de la ejecución de las mismas y por tanto de la comunicación con el módulo y de este con el servidor, informado de los posibles errores por la salida estándar.

Cierra el dispositivo especial de caracteres por medio de la llamada al sistema:

```
result = close(file);
```

Por último libera la memoria reservada y termina la ejecución.

A continuación se muestra la forma de invocar las *ioctls*:

```
resultado = ioctl(file, comando, parametros);
```

Donde:

- ***File***: es el descriptor de fichero del dispositivo especial de caracteres.
- ***Comando***: número que identifica de forma univoca las distintas ioctls.
- ***Parámetros***: estructura que contiene los parámetros necesarios para llevar a cabo la petición solicitada por el cliente.

Todas devuelven valor cero en caso de éxito, cualquier otro valor indica que se ha producido un error.

Declaración de la estructura parámetros:

```
struct parametros {
      struct sockaddr_in    dir_ser;
      char                  *path;
      char                  *name;
      char                  link;
      fhstatus              fhs;
      fhstatus              *fhin;
      fhstatus              *fhout;
      struct nfs_fattr      *fattr
      char                  *buffer;
      long                  size;
      unsigned int          permisos;
};
typedef struct parametros  parametros;
```

Donde:

- ***Dir_ser***: Es el socket para la comunicación con el servidor, contiene la dirección del mismo.
- ***Path***: Contiene, en caso necesario, un path para la petición que se desea realizar.
- ***Name***: Contiene, en caso necesario, un nombre de un objeto del sistema de ficheros.
- ***Link***: Contiene el path, al que apunta el enlace simbólico, obtenido en la petición readlink en caso de éxito.
- ***Fhs***: Contiene el filehandle, manejador de fichero, inicial obtenido en la petición mount en caso de éxito.
- ***Fhin***: Contiene un filehandle, en caso necesario, para la petición que se desea realizar.
- ***Fhout***: Contiene un filehandle obtenido como resultado de la petición realizada en caso de éxito.
- ***Fattr***: Contiene los atributos de un objeto del sistema de ficheros en caso necesario para realizar la petición o como resultado de una realizada en caso de éxito.
- ***Buffer***: Contiene los datos obtenidos o necesarios para las peticiones de lectura o escritura.
- ***Size***: Contiene, en caso necesario, el tamaño de los datos de lectura o escritura.
- ***Permisos***: Contiene, en caso necesario, los permisos del objeto del sistema de ficheros.

7.1.1 Protocolo mount

Este apartado muestra la forma de invocar las *ioctls* del protocolo de montado.

7.1.1.1 null

```
resultado = ioctl(file, 0x101, &param);
```

Invocación de la *ioctl null* que es recogida por el módulo del kernel, mi_nfs. Pasa como parámetro la estructura *parámetros*, la cual contiene la dirección de la máquina servidor para hacerle llegar la petición por medio de la rpc_call del procedimiento.

7.1.1.2 mount

```
resultado = ioctl(file, 0x102, &param);
```

Invocación de la *ioctl mount* que es recogida por el módulo del kernel, mi_nfs. Pasa como parámetro la estructura *parámetros*, la cual contiene el path que se desea montar y la dirección de la máquina servidor para hacerle llegar la petición por medio de la rpc_call del procedimiento.

7.1.1.3 dump

```
resultado = ioctl(file, 0x103, &param);
```

Invocación de la *ioctl dump* que es recogida por el módulo del kernel, mi_nfs. Pasa como parámetro la estructura *parámetros*, la cual contiene la dirección de la máquina servidor para hacerle llegar la petición por medio de la rpc_call del procedimiento.

7.1.1.4 umount

```
resultado = ioctl(file, 0x104, &param);
```

Invocación de la *ioctl umount* que es recogida por el módulo del kernel, mi_nfs. Pasa como parámetro la estructura *parámetros*, la cual contiene el path que se desea desmontar y la dirección de la máquina servidor para hacerle llegar la petición por medio de la rpc_call del procedimiento.

7.1.1.5 umountall

```
resultado = ioctl(file, 0x105, &param);
```

Invocación de la *ioctl umountall* que es recogida por el módulo del kernel, mi_nfs. Pasa como parámetro la estructura *parámetros*, la cual contiene la dirección de la máquina servidor para hacerle llegar la petición por medio de la rpc_call del procedimiento.

7.1.1.6 export

```
resultado = ioctl(file, 0x106, &param);
```

Invocación de la *ioctl export* que es recogida por el módulo del kernel, mi_nfs. Pasa como parámetro la estructura *parámetros*, la cual contiene la dirección de la máquina servidor para hacerle llegar la petición por medio de la rpc_call del procedimiento.

7.1.2 Protocolo NFS

Este apartado muestra la forma de invocar las ioctls del protocolo NFS.

7.1.2.1 null

```
resultado = ioctl(file, 0x107, &param);
```

Invocación de la *ioctl null* que es recogida por el módulo del kernel, mi_nfs. Pasa como parámetro la estructura *parámetros*, la cual contiene la dirección de la máquina servidor para hacerle llegar la petición por medio de la rpc_call del procedimiento.

7.1.2.2 getattr

```
resultado = ioctl(file, 0x108, &param);
```

Invocación de la *ioctl getattr* que es recogida por el módulo del kernel, mi_nfs. Pasa como parámetro la estructura *parámetros*, la cual contiene el *filehandle*, manejador de fichero, del path del que se desea obtener los atributos y la dirección de la máquina servidor para hacerle llegar la petición por medio de la rpc_call del procedimiento. Recibe los atributos solicitados en la estructura *nfs_fattr* de la estructura *parámetros*.

7.1.2.3 setattr

```
resultado = ioctl(file, 0x109, &param);
```

Invocación de la *ioctl setattr* que es recogida por el módulo del kernel, mi_nfs. Pasa como parámetro la estructura *parámetros*, la cual contiene el *filehandle*, manejador de fichero, del path del que se desea cambiar los atributos, los atributos que se desean modificar en la estructura *nfs_fattr* y la dirección de la máquina servidor para hacerle llegar la petición por medio de la rpc_call del procedimiento. Recibe los atributos cambiados en la estructura *nfs_fattr* de la estructura *parámetros*.

7.1.2.4 lookup

```
resultado = ioctl(file, 0x10a, &param);
```

Invocación de la *ioctl lookup* que es recogida por el módulo del kernel, mi_nfs. Pasa como parámetro la estructura *parámetros*, la cual contiene el path, nombre del fichero o directorio al que se desea acceder, el *filehandle*, manejador de fichero, del directorio donde se encuentra dicho path y la dirección de la máquina servidor para hacerle llegar la petición por medio de la rpc_call del procedimiento. Recibe el *filehandle*, manejador de fichero, del path en *fhout* y los atributos en la estructura *nfs_fattr* de la estructura *parámetros*.

7.1.2.5 readlink

```
resultado = ioctl(file, 0x10b, &param);
```

Invocación de la *ioctl readlink* que es recogida por el módulo del kernel, mi_nfs. Pasa como parámetro la estructura *parámetros*, la cual contiene el *filehandle*, manejador de fichero, del enlace simbólico que se desea leer y la dirección de la máquina servidor para hacerle llegar la petición por medio de la rpc_call del procedimiento. Recibe el path al que apunta el enlace simbólico en el campo *link* de la estructura *parámetros*.

7.1.2.6 read

```
resultado = ioctl(file, 0x10c, &param);
```

Invocación de la *ioctl read* que es recogida por el módulo del kernel, mi_nfs. Pasa como parámetro la estructura *parámetros*, la cual contiene el *filehandle*, manejador de fichero, que se desea leer, el tamaño de bloque y el tamaño en la estructura *nfs_fattr* y la dirección de la máquina servidor para hacerle llegar la petición por medio de la rpc_call del procedimiento. Recibe el contenido del fichero solicitado en el campo buffer de la estructura *parámetros*.

7.1.2.7 write

```
resultado = ioctl(file, 0x10d, &param);
```

Invocación de la *ioctl write* que es recogida por el módulo del kernel, mi_nfs. Pasa como parámetro la estructura *parámetros*, la cual contiene el *filehandle*, manejador de fichero, en el que se va a escribir, el tamaño de bloque y el tamaño en la estructura *nfs_fattr*, el buffer con el contenido que se desea escribir y la dirección de la máquina servidor para hacerle llegar la petición por medio de la *rpc_call* del procedimiento. Recibe los atributos del fichero modificado en la estructura *nfs_fattr* de la estructura *parámetros*.

7.1.2.8 create

```
resultado = ioctl(file, 0x10e, &param);
```

Invocación de la *ioctl create* que es recogida por el módulo del kernel, mi_nfs. Pasa como parámetro la estructura *parámetros*, la cual contiene el *filehandle*, manejador de directorio, en el que se va a crear el fichero, los permisos que va a tener, el nombre del fichero a crear y la dirección de la máquina servidor para hacerle llegar la petición por medio de la rpc_call del procedimiento. Recibe en la estructura *parámetros* el *filehandle*, manejador del fichero creado en el campo *fhout* y los atributos del fichero en la estructura *nfs_fattr*.

7.1.2.9 remove

```
resultado = ioctl(file, 0x10f, &param);
```

Invocación de la *ioctl remove* que es recogida por el módulo del kernel, mi_nfs. Pasa como parámetro la estructura *parámetros*, la cual contiene el *filehandle*, manejador de directorio, en el que se encuentra el fichero a borrar, el nombre del fichero a borrar y la dirección de la máquina servidor para hacerle llegar la petición por medio de la rpc_call del procedimiento.

7.1.2.10 rename

```
resultado = ioctl(file, 0x110, &param);
```

Invocación de la *ioctl rename* que es recogida por el módulo del kernel, mi_nfs. Pasa como parámetro la estructura *parámetros*, la cual contiene el *filehandle*, manejador del fichero o directorio, en el que se encuentra el que se quiere renombrar, el nombre de este, el nuevo nombre que va a tener y la dirección de la máquina servidor para hacerle llegar la petición por medio de la rpc_call del procedimiento.

7.1.2.11 link

```
resultado = ioctl(file, 0x111, &param);
```

Invocación de la *ioctl link* que es recogida por el módulo del kernel, mi_nfs. Pasa como parámetro la estructura *parámetros*, la cual contiene el *filehandle*, manejador del directorio, en el que vamos a crear el enlace físico, el nombre de este, el *filehandle*, manejador del directorio, al que apunta el enlace físico y la dirección de la máquina servidor para hacerle llegar la petición por medio de la rpc_call del procedimiento.

7.1.2.12 symlink

```
resultado = ioctl(file, 0x112, &param);
```

Invocación de la *ioctl symlink* que es recogida por el módulo del kernel, mi_nfs. Pasa como parámetro la estructura *parámetros*, la cual contiene el *filehandle*, manejador del directorio, en el que vamos a crear el enlace simbólico, el nombre de este, el path al que apunta, los permisos y la dirección de la máquina servidor para hacerle llegar la petición por medio de la rpc_call del procedimiento.

7.1.2.13 mkdir

```
resultado = ioctl(file, 0x113, &param);
```

Invocación de la *ioctl mkdir* que es recogida por el módulo del kernel, mi_nfs. Pasa como parámetro la estructura *parámetros*, la cual contiene el *filehandle*, manejador del directorio, en el que vamos a crear el nuevo directorio, el nombre de este, los permisos y la dirección de la máquina servidor para hacerle llegar la petición por medio de la rpc_call del procedimiento. Recibe en la estructura *parámetros* el *filehandle*, manejador del directorio creado en el campo *fhout* y los atributos en la estructura *nfs_fattr*.

7.1.2.14 rmdir

```
resultado = ioctl(file, 0x114, &param);
```

Invocación de la *ioctl rmdir* que es recogida por el módulo del kernel, mi_nfs. Pasa como parámetro la estructura *parámetros*, la cual contiene el *filehandle*, manejador de directorio, en el que se encuentra el directorio a borrar, el nombre del directorio a borrar y la dirección de la máquina servidor para hacerle llegar la petición por medio de la rpc_call del procedimiento.

7.1.2.15 statfs

```
resultado = ioctl(file, 0x115, &param);
```

Invocación de la *ioctl statfs* que es recogida por el módulo del kernel, mi_nfs. Pasa como parámetro la estructura *parámetros*, la cual contiene el *filehandle*, manejador de fichero o directorio, del que vamos a obtener los atributos del sistema de ficheros que le contiene y la dirección de la máquina servidor para hacerle llegar la petición por medio de la rpc_call del procedimiento.

7.2 Acceso al módulo mediante ioctl

En este apartado se describe y muestra la declaración de los distintos comandos u órdenes existentes en el módulo cliente de NFS. En el capítulo anterior se ha explicado todo lo relativo a la función especial ioctl, véase el apartado 6.7.2.1 ioctl.

La función especial *ioctl* estará definida por las siguientes órdenes:

```
#define MI_NFS_IOC_NMAGICO   1
```

El número, orden o comando, *ioctl* es creado utilizando la macro de la librería *asm/ioctl.h*:

```
// Para el protocolo MOUNT:
// Comando del nulo
#define MI_NFS_IOC_MNT_NULO  _IO(MI_NFS_IOC_NMAGICO, 1)
// Comando del mount
#define MI_NFS_IOC_MOUNT     _IO(MI_NFS_IOC_NMAGICO, 2)
// Comando del dump
#define MI_NFS_IOC_DUMP      _IO(MI_NFS_IOC_NMAGICO, 3)
// Comando del umount
#define MI_NFS_IOC_UMNT      _IO(MI_NFS_IOC_NMAGICO, 4)
// Comando del umountall
#define MI_NFS_IOC_UMNTALL   _IO(MI_NFS_IOC_NMAGICO, 5)
// Comando del export
#define MI_NFS_IOC_EXPORT    _IO(MI_NFS_IOC_NMAGICO, 6)

// Para el protocolo NFS:
// Comando del nulo
#define MI_NFS_IOC_NULO      _IO(MI_NFS_IOC_NMAGICO, 7)
// Comando del getattr
#define MI_NFS_IOC_GETATTR   _IO(MI_NFS_IOC_NMAGICO, 8)
// Comando del setattr
#define MI_NFS_IOC_SETATTR   _IO(MI_NFS_IOC_NMAGICO, 9)
// Comando del lookup
#define MI_NFS_IOC_LOOKUP    _IO(MI_NFS_IOC_NMAGICO, 10)
// Comando del readlink
#define MI_NFS_IOC_READLINK  _IO(MI_NFS_IOC_NMAGICO, 11)
// Comando del read
#define MI_NFS_IOC_READ      _IO(MI_NFS_IOC_NMAGICO, 12)
// Comando del write
#define MI_NFS_IOC_WRITE     _IO(MI_NFS_IOC_NMAGICO, 13)
// Comando del create
#define MI_NFS_IOC_CREATE    _IO(MI_NFS_IOC_NMAGICO, 14)
// Comando del remove
#define MI_NFS_IOC_REMOVE    _IO(MI_NFS_IOC_NMAGICO, 15)
// Comando del rename
#define MI_NFS_IOC_RENAME    _IO(MI_NFS_IOC_NMAGICO, 16)
// Comando del link
#define MI_NFS_IOC_LINK      _IO(MI_NFS_IOC_NMAGICO, 17)
// Comando del symlink
#define MI_NFS_IOC_SYMLINK   _IO(MI_NFS_IOC_NMAGICO, 18)
// Comando del mkdir
```

```
#define MI_NFS_IOC_MKDIR     _IO(MI_NFS_IOC_NMAGICO, 19)
// Comando del rmdir
#define MI_NFS_IOC_RMDIR     _IO(MI_NFS_IOC_NMAGICO, 20)
// Comando del statfs
#define MI_NFS_IOC_STATFS    _IO(MI_NFS_IOC_NMAGICO, 21)
```

El número de ioctls que he definido:

```
#define MI_NFS_IOC_NUM_MAX  21
```

Cuando se realiza una llamada *ioctl* se invoca la siguiente función del módulo, cuya cabecera está definida de la siguiente forma:

```
int mi_nfs_ioctl(struct inode *pinodo, struct file
*pfile, unsigned int comando, struct parametros *param)
```

Donde:

- ***Inode***: es un puntero a una estructura de tipo inodo la cual contiene, entre otras cosas, un puntero a la estructura *inode_operations*, a *file_operations* y a *super_block*, así como el tipo de sistema de ficheros del inodo, etc. Definida en *include/linux/fs.h*.
- ***File***: puntero a la estructura de tipo file definida en *include/linux/fs.h*. Contiene entre otras cosas un puntero a la estructura *dentry*, *vfsmount* y *file_operations*, etc.
- ***Comando***: parámetro numérico que identifica la orden a realizar por el módulo del núcleo.
- ***Param***: puntero a la estructura parámetros que contiene los parámetros necesarios para llevar a cabo la operación.

mi_nfs_ioctl() es la función del módulo que recibe las llamadas *ioctls* con sus correspondientes parámetros. Realiza las comprobaciones pertinentes del comando recibido, tales como que el *type* y el *number*, número que identifica cada acción, sean correctos. Se especifica el protocolo a utilizar, TCP o UDP. En función del número de la *ioctl* invocada se identifica el servicio, NFS o MOUNT, y se obtiene el número de puerto en el que el servidor espera peticiones, por medio de la función *obtener_puerto (número de programa, versión, protocolo, máquina servidor)*. Prepara la comunicación por medio de la función *prepara_sockaddr()*, determina a través del comando recibido en la llamada ioctl la acción a realizar, y llama a la función encargada de tratar la petición, por ejemplo *mi_mount()*, la cual se encarga de crear la conexión con el servidor y enviarle la petición, *rpc_call()*, poniendo los datos necesarios para la realización de la misma en formato de red. Por último recibe los datos del servidor poniéndolos en formato máquina, controla y devuelve el estado de la ejecución en el servidor para informar del mismo.

Pseudocódigo:

```
int mi_nfs_ioctl()
{
  if (_IOC_TYPE(comando) != MI_NFS_IOC_NMAGICO)
     return -EINVAL;
  if (_IOC_NR(comando) > MI_NFS_IOC_NUM_MAX)
     return -EINVAL;
  proto = IPPROTO_UDP;
```

```
if (_IOC_NR(comando) < 7)
    puerto      =      obtener_puerto(MI_MOUNT_PROGRAM,
    MI_MOUNT_VERS, proto, dir_ser);
else
    puerto      =       obtener_puerto(MI_NFS_PROGRAM,
    MI_NFS_VERS, proto, dir_ser);

prepara_sockaddr(&sin, dir_ser.sin_addr.s_addr, 0);

switch(comando){
  case MI_NFS_IOC_MNT_NULO:
    res = mi_mount_nulo(&sin, proto, puerto);
    break;
  case MI_NFS_IOC_MOUNT:
    res = mi_mount(&sin, path, proto, puerto, &fhs);
    break;
  case MI_NFS_IOC_DUMP:
    res = mi_dump(&sin, proto, puerto);
    break;
  case MI_NFS_IOC_UMNT:
    res = mi_umnt(&sin, proto, puerto, path);
    break;
  case MI_NFS_IOC_UMNTALL:
    res = mi_umntall(&sin, proto, puerto);
    break;
  case MI_NFS_IOC_EXPORT:
    res = mi_export(&sin, proto, puerto);
    break;
  case MI_NFS_IOC_NULO:
    res = mi_nulo(&sin, proto, puerto, nfs_nulo);
    break;
  case MI_NFS_IOC_GETATTR:
    res = mi_getattr(&sin, proto, puerto, fhin, fattr);
    break;
  case MI_NFS_IOC_SETATTR:
    res = mi_setattr(&sin, proto, puerto, fhin, fattr);
    break;
  case MI_NFS_IOC_LOOKUP:
    res  =  mi_lookup(&sin,  path,  proto,  puerto,  fhin,
    fattr, fhout);
    break;
  case MI_NFS_IOC_READLINK:
    res   =   mi_readlink(&sin,   proto,   puerto,   &fhin-
    >fhstatus_u.directory, &link);
    break;
  case MI_NFS_IOC_READ:
    res    =    mi_read(&sin,    proto,    puerto,    &fhin-
    >fhstatus_u.directory,    offset,    buffer,    size,
    blockSize);
    break;
  case MI_NFS_IOC_WRITE:
```

```
        res    =   mi_write(&sin,   proto,   puerto,   &fhin-
        >fhstatus_u.directory,   buffer,   offset,   size,
        blockSize, fattr);
        break;
      case MI_NFS_IOC_CREATE:
        res  = mi_create(&sin, path, proto, puerto, fhin,
        permisos, fattr, fhout);
        break;
      case MI_NFS_IOC_REMOVE:
        res  = mi_remove(&sin, path, proto, puerto, fhin-
        >fhstatus_u.directory);
        break;
      case MI_NFS_IOC_RENAME:
        res  = mi_rename(&sin, proto, puerto, path, fhin-
        >fhstatus_u.directory, name);
        break;
      case MI_NFS_IOC_LINK:
        res    =   mi_link(&sin,   proto,   puerto,   fhout-
        >fhstatus_u.directory,  fhin->fhstatus_u.directory,
        path);
        break;
      case MI_NFS_IOC_SYMLINK:
        res   =   mi_symlink(&sin,   proto,   puerto,   fhin-
        >fhstatus_u.directory, name, path, permisos);
        break;
      case MI_NFS_IOC_MKDIR:
        res   =   mi_mkdir   (&sin,   proto,   puerto,   fhin-
        >fhstatus_u.directory,   path,   permisos,   fhout-
        >fhstatus_u.directory, fattr);
        break;
      case MI_NFS_IOC_RMDIR:
        res    =   mi_rmdir(&sin,   proto,   puerto,   fhin-
        >fhstatus_u.directory, path);
        break;
      case MI_NFS_IOC_STATFS:
        res    =   mi_statfs(&sin,   proto,   puerto,   fhin-
        >fhstatus_u.directory);
        break;
      default:
        return -EINVAL;
        break;
    }
    return res;
  }
```

7.3 Implementación de las funciones

En este apartado se describe y muestra la implementación de las funciones del módulo encargadas de llevar a cabo las correspondientes peticiones al servidor, así como de recoger los resultados devueltos por el mismo y retornarlos a la función principal de las *ioctls*, *mi_nfs_ioctl()*.

7.3.1 Portmapper

A continuación se describen cada una de las funciones del servidor portmapper utilizadas en la implementación del módulo cliente de NFS versión 2.

7.3.1.1 Obtener puerto

Recibe como parámetros de entrada el número de programa, el número de versión, el protocolo de comunicación, y el socket para la comunicación. Esta función prepara la conexión con el servidor portmapper, *prepara_sockaddr()*, e invoca la función *rpc_getport_external()*, que es la encargada de solicitar la petición para obtener el puerto y devolverlo como parámetro de salida.

```
int obtener_puerto(int program, int version, int proto,
struct sockaddr_in dir_ser)
{
	struct sockaddr_in sin;

	prepara_sockaddr(&sin, dir_ser.sin_addr.s_addr, 0);
	return rpc_getport_external(&sin, program, version,
	proto);
}
```

7.3.1.1.1 Rpc_getport_external

Recibe como parámetros de entrada el socket, el número de programa, el número de versión y el protocolo. Introduce en la estructura *rpc_portmap* el programa, la versión, el protocolo y la respuesta, inicialmente valor cero. Crea un cliente RPC para el portmap, *pmap_clnt*, por medio de la función *pmap_create()* y realiza la petición al servidor, *rpc_call()*, recibe la respuesta y la devuelve como parámetro de salida.

```
int rpc_getport_external(struct sockaddr_in *sin, __u32
prog, __u32 vers, int proto)
{
	struct rpc_portmap map = { prog, vers, proto, 0 };
	struct rpc_clnt     *pmap_clnt;
	char                hostname[32];
	int                 status;

	strcpy(hostname, in_ntoa(sin->sin_addr.s_addr));
	if  (!(pmap_clnt  =  pmap_create(hostname,  sin,
	proto)))
		return -EACCES;

	status  =  rpc_call(pmap_clnt,  PMAP_GETPORT,  &map,
	&map.pm_port, 0);

	if (status >= 0) {
		if (map.pm_port != 0)
			return map.pm_port;
		status = -EACCES;
	}
	return status;
```

```
}
```

7.3.1.1.2 Pmap_create

Recibe como parametros de entrada el nombre de la maquina servidor, el socket y el protocolo. Crea un cliente de transporte de RPC, dado el par protocolo - dirección de la máquina, por medio de la llamada al sistema *xprt_create_proto()*, definida en */net/sunrpc/xprt.c.* Crea un cliente RPC por medio de la llamada al sistema *rpc_create_client()*, definida en */net/sunrpc/clnt.c*, y lo devuelve como argumento de salida.

```
struct rpc_clnt * pmap_create(char *hostname, struct
sockaddr_in *srvaddr, int proto)
{
	struct rpc_xprt *xprt;
	struct rpc_clnt *clnt;

	if (!(xprt = xprt_create_proto(proto, srvaddr,
	NULL)))
		return NULL;

	xprt->addr.sin_port = htons(RPC_PMAP_PORT);
	clnt = rpc_create_client(xprt, hostname,
	&pmap_program, RPC_PMAP_VERSION, RPC_AUTH_NULL);

	if (!clnt) {
		xprt_destroy(xprt);
	} else {
		clnt->cl_softrtry = 1;
		clnt->cl_chatty   = 1;
		clnt->cl_oneshot  = 1;
	}
	return clnt;
}
```

7.3.2 Protocolo mount

A continuación se describen cada una de las funciones del servidor mount utilizadas en la implementación del módulo cliente de NFS versión 2.

7.3.2.1 Mi_mount_create

Recibe como parametros de entrada el nombre de la máquina servidor, el socket, el protocolo y el puerto. Crea un cliente de transporte de RPC, dado el par protocolo - dirección de la máquina, por medio de la llamada al sistema *xprt_create_proto()*, definida en */net/sunrpc/xprt.c.* Crea un cliente RPC por medio de la llamada al sistema *rpc_create_client()*, definida en */net/sunrpc/clnt.c*, y lo devuelve como argumento de salida.

```
struct rpc_clnt * mi_mount_create(char *hostname, struct
sockaddr_in *srvaddr, int proto, int puerto)
{
	struct rpc_xprt *xprt;
	struct rpc_clnt *clnt;
```

```
        if (!(xprt = xprt_create_proto(proto, srvaddr, NULL)))
              return NULL;

        xprt->addr.sin_port = htons(puerto);
        clnt = rpc_create_client(xprt, hostname, &mi_mount_program, MI_MOUNT_VERS, RPC_AUTH_NULL);

        if (!clnt) {
              xprt_destroy(xprt);
        } else {
              clnt->cl_softrtry = 1;
              clnt->cl_chatty   = 1;
              clnt->cl_oneshot  = 1;
        }
        return clnt;
}
```

7.3.2.2 Nulo

Recibe como parámetros de entrada el socket, el protocolo y el puerto. Crea un cliente RPC para el mount, *mi_nfs_clnt*, por medio de la función *mi_mount_create()* y realiza la petición al servidor, *rpc_call().* Recibe la respuesta devuelta por el servidor, cero en caso de éxito, y la devuelve como parámetro de salida.

```
int mi_mount_nulo(struct sockaddr_in *sin, int proto, int puerto)
{
        struct rpc_clnt  *mi_nfs_clnt;
        void             *argp;
        void             *clnt_res;
        char             hostname[FHSIZE];
        int              estado = 0;

        strcpy(hostname, in_ntoa(sin->sin_addr.s_addr));
        if (!(mi_nfs_clnt = mi_mount_create(hostname, sin, proto, puerto)))
              return -EACCES;

        estado = rpc_call(mi_nfs_clnt, nulo, argp, clnt_res, 0);
        return estado;
}
```

7.3.2.3 Mount

Esta función obtiene el manejador o filehandle inicial. Se obtiene montando el directorio pasado por parámetro. Recibe como parámetros de entrada el socket, el path absoluto del cual se desea obtener el manejador, el protocolo y el puerto. Crea un cliente RPC para el mount, *mi_nfs_clnt*, por medio de la función *mi_mount_create()* y realiza la petición al servidor, *rpc_call().* Recibe la respuesta, en la variable *clnt_res*, y el estado, cero en caso de éxito, devueltos por el servidor y los devuelve como parámetro de salida, *clnt_res*.

```
int mi_mount (struct sockaddr_in *sin, char *path, int
proto, int puerto, struct fhstatus *clnt_res)
{
      struct rpc_clnt *mi_nfs_clnt;
      char            hostname[FHSIZE];

      strcpy(hostname, in_ntoa(sin->sin_addr.s_addr));
      if (!(mi_nfs_clnt = mi_mount_create(hostname, sin,
      proto, puerto)))
            return -EACCES;

      clnt_res->status = rpc_call(mi_nfs_clnt, montar,
      path, clnt_res, 0);
      return clnt_res->status;
}
```

7.3.2.3 Export

Esta función obtiene todos los directorios exportados por un servidor. Recibe como parámetros de entrada el socket, el protocolo y el puerto. Crea un cliente RPC para el mount, *mi_nfs_clnt*, por medio de la función *mi_mount_create()* y realiza la petición al servidor, *rpc_call().* Recibe la respuesta, en la variable *exp*, y el estado, cero en caso de éxito, devueltos por el servidor y devuelve este último como parámetro de salida.

Exp es una variable de la estructura *exportnode*, que contiene una lista de los directorios exportados por el servidor, junto a los permisos asociados al directorio.

```
int mi_export(struct sockaddr_in *sin, int proto, int
puerto)
{
      struct rpc_clnt       *mi_nfs_clnt;
      char                  hostname[FHSIZE];
      int                   estado = 0;
      void                  *argp;
      struct exportnode     exp;

      strcpy(hostname, in_ntoa(sin->sin_addr.s_addr));
      if (!(mi_nfs_clnt = mi_mount_create(hostname, sin,
      proto, puerto)))
            return -EACCES;

      estado = rpc_call(mi_nfs_clnt, exportar, argp,
      &exp, 0);
      return estado;
}
```

7.3.2.4 Dump

Obtiene del servidor una lista de *paths* y máquinas montados. Recibe como parámetros de entrada el socket, el protocolo y el puerto. Crea un cliente RPC para el mount, *mi_nfs_clnt*, por medio de la función *mi_mount_create()* y realiza la petición al servidor, *rpc_call().* Recibe la

respuesta, en la variable *clnt_res*, y el estado, cero en caso de éxito, devueltos por el servidor y devuelve este último como parámetro de salida.

Clnt_res es una variable de la estructura *mountlist* que contiene una lista de *paths* y máquinas montados.

```
int mi_dump(struct sockaddr_in *sin, int proto, int
puerto)
{
      struct rpc_clnt *mi_nfs_clnt;
      char            hostname[FHSIZE];
      int             estado = 0;
      void            *argp;
      mountlist       clnt_res;

      strcpy(hostname, in_ntoa(sin->sin_addr.s_addr));
      if (!(mi_nfs_clnt = mi_mount_create(hostname, sin,
      proto, puerto)))
      return -EACCES;

      estado = rpc_call(mi_nfs_clnt, dump, argp,
      &clnt_res, 0);
      return estado;
}
```

7.3.2.5 Umount

Elimina la entrada en la tabla de *path's* montados en el servidor. Recibe como parámetros de entrada el socket, el protocolo, el puerto y el path absoluto de un directorio montado. Crea un cliente RPC para el mount, *mi_nfs_clnt*, por medio de la función *mi_mount_create()* y realiza la petición al servidor, *rpc_call().* Recibe la respuesta, en la variable *clnt_res*, y el estado, cero en caso de éxito, devueltos por el servidor y devuelve este último como parámetro de salida.

```
int mi_umnt(struct sockaddr_in *sin, int proto, int
puerto, char *path)
{
      struct rpc_clnt *mi_nfs_clnt;
      char            hostname[FHSIZE];
      int             estado = 0;
      void            *clnt_res;

      strcpy(hostname, in_ntoa(sin->sin_addr.s_addr));
      if (!(mi_nfs_clnt = mi_mount_create(hostname, sin,
      proto, puerto)))
            return -EACCES;

      estado = rpc_call(mi_nfs_clnt, umnt, path,
      clnt_res, 0);
      return estado;
}
```

7.3.2.6 Umountall

Esta función elimina todas las entradas en la tabla de *path's* montados en el servidor. Recibe como parámetros de entrada el socket, el protocolo y el puerto. Crea un cliente RPC para el mount, *mi_nfs_clnt*, por medio de la función *mi_mount_create()* y realiza la petición al servidor, *rpc_call().* Recibe la respuesta, en la variable *clnt_res*, y el estado, cero en caso de éxito, devueltos por el servidor y devuelve este último como parámetro de salida.

```
int mi_umntall(struct sockaddr_in *sin, int proto, int
puerto)
{
      struct rpc_clnt  *mi_nfs_clnt;
      char             hostname[FHSIZE];
      int              estado = 0;
      void             *argp;
      void             *clnt_res;


      strcpy(hostname, in_ntoa(sin->sin_addr.s_addr));
      if  (!(mi_nfs_clnt  = mi_mount_create(hostname,  sin,
      proto, puerto)))
            return -EACCES;

      estado   =   rpc_call(mi_nfs_clnt,   umntall,   argp,
      clnt_res, 0);
      return estado;
}
```

7.3.3 Protocolo NFS

A continuación se describen cada una de las funciones del servidor NFS utilizadas en la implementación del módulo cliente de NFS versión 2.

7.3.2.1 Mi_nfs_create

Recibe como parametros de entrada el nombre de la máquina servidor, el socket, el protocolo y el puerto. Crea un cliente de transporte de RPC, dado el par protocolo - dirección de la máquina, por medio de la llamada al sistema *xprt_create_proto()*, definida en */net/sunrpc/xprt.c.* Crea un cliente RPC por medio de la llamada al sistema *rpc_create_client()*, definida en */net/sunrpc/clnt.c*, y lo devuelve como argumento de salida.

```
struct  rpc_clnt  *  mi_nfs_create(char  *hostname,  struct
sockaddr_in *srvaddr, int proto, int puerto)
{
      struct rpc_xprt  *xprt;
      struct rpc_clnt  *clnt;

      if   (!(xprt   =   xprt_create_proto(proto,   srvaddr,
      NULL)))
            return NULL;

      xprt->addr.sin_port = htons(puerto);
```

```
        clnt     =     rpc_create_client(xprt,     hostname,
        &mi_nfs_program, MI_NFS_VERS, RPC_AUTH_NULL);

        if (!clnt) {
              xprt_destroy(xprt);
        } else {
              clnt->cl_softrtry = 1;
              clnt->cl_chatty   = 1;
              clnt->cl_oneshot  = 1;
        }
        return clnt;
}
```

7.3.2.2 Nulo

Recibe como parámetros de entrada el socket, el protocolo, el puerto y un entero que identifica la llamada *nulo, root* o *writecache*. Crea un cliente RPC para el mount, *mi_nfs_clnt*, por medio de la función *mi_nfs_create()* y realiza la petición al servidor, *rpc_call().* Recibe la respuesta devuelta por el servidor, cero en caso de éxito, y la devuelve como parámetro de salida.

```
int  mi_nulo(struct  sockaddr_in  *sin,  int  proto,  int
puerto, int llamada)
{
        struct rpc_clnt  *mi_nfs_clnt;
        void             *argp;
        void             *clnt_res;
        char             hostname[FHSIZE];
        int              estado = 0;

        strcpy(hostname, in_ntoa(sin->sin_addr.s_addr));
        if  (!(mi_nfs_clnt  =  mi_nfs_create(hostname,  sin,
        proto, puerto)))
              return -EACCES;

        estado   =  rpc_call(mi_nfs_clnt,   llamada,   argp,
        clnt_res, 0);
        return estado;
}
```

7.3.2.3 Getattr

Esta función obtiene los atributos de un objeto del sistema de ficheros del que se tiene el manejador. Recibe como parámetros de entrada el socket, el protocolo, el puerto, el manejador del que se quiere obtener los atributos y un puntero a una estructura con los atributos del fichero, se rellena en caso de éxito. Crea un cliente RPC para el mount, *mi_nfs_clnt*, por medio de la función *mi_nfs_create()* y realiza la petición al servidor, *rpc_call().* Recibe los atributos del objeto en la variable *fattr* y el estado devueltos por el servidor, cero en caso de éxito, y los devuelve como parámetros de salida.

```
int mi_getattr(struct sockaddr_in *sin, int proto, int
puerto, struct fhstatus *fh, struct nfs_fattr *fattr)
{
      struct rpc_clnt   *mi_nfs_clnt;
      char              hostname[FHSIZE];
      int               estado = 0;

      strcpy(hostname, in_ntoa(sin->sin_addr.s_addr));
      if (!(mi_nfs_clnt = mi_nfs_create(hostname, sin,
      proto, puerto)))
      return -EACCES;

      if((estado = rpc_call(mi_nfs_clnt, nfs_getattr, fh-
      >fhstatus_u.directory, fattr, 0)) == NFS_OK){
            printk("mi_nfs> NFS> RPC> Atributos:\n");
            // Mensajes por pantalla con los atributos
            del objeto del sistema de ficheros
      }
      return estado;
}
```

7.3.2.4 Setattr

Esta función cambia los atributos a un objeto del sistema de ficheros del que se tiene el manejador. Recibe como parámetros de entrada el socket, el protocolo, el puerto, el manejador del que se quiere cambiar los atributos y un puntero a una estructura con los atributos nuevos del fichero, se rellena en caso de éxito. Crea un cliente RPC para el mount, *mi_nfs_clnt*, por medio de la función *mi_nfs_create()* y realiza la petición al servidor, *rpc_call().* Recibe los atributos modificados del objeto en *fattr* y el estado devueltos por el servidor, cero en caso de éxito, y los devuelve como parámetros de salida.

Sattr es una variable de la estructura *iattr* que contiene los valores de entrada para el cambio de atributos.

```
int mi_setattr(struct sockaddr_in *sin, int proto, int
puerto, struct fhstatus *fh, struct nfs_fattr *fattr)
{
      struct rpc_clnt   *mi_nfs_clnt;
      char              hostname[FHSIZE];
      int               estado = 0;
      struct iattr      sattr;

      strcpy(hostname, in_ntoa(sin->sin_addr.s_addr));
      if (!(mi_nfs_clnt = mi_nfs_create(hostname, sin,
      proto, puerto)))
            return -EACCES;

      sattr.ia_mode           = fattr->mode;
      sattr.ia_uid            = fattr->uid;
      sattr.ia_gid            = fattr->gid;
      sattr.ia_size           = fattr->size;
```

```
		memcpy(&sattr.ia_atime,		&(fattr->atime),
		sizeof(__u64));
		memcpy(&sattr.ia_mtime,		&(fattr->mtime),
		sizeof(__u64));

		fattr->valid = 0;

		if((estado  =  nfs_proc_setattr(mi_nfs_clnt,  &fh-
		>fhstatus_u.directory, &sattr, fattr)) == NFS_OK){
			printk("mi_nfs> NFS> RPC> Atributos:\n");
			// Mensajes por pantalla con los atributos
			del objeto del sistema de ficheros
		}
		return estado;
	}

	static    int    nfs_proc_setattr(struct    rpc_clnt
	*mi_nfs_clnt, fhandle *fh, struct iattr *sattr, struct
	nfs_fattr *fattr)
	{
		struct sattrargs	arg = { fh, sattr };
		int			status;

		fattr->valid = 0;
		status  =  rpc_call(mi_nfs_clnt,  NFSPROC_SETATTR,
		&arg, fattr, 0);
		return status;
	}
```

7.3.2.5 Lookup

Esta función obtiene el manejador de un *path*. Recibe como parámetros de entrada el *socket*, el *path* del que queremos obtener el *filehandle*, el protocolo, el puerto, el manejador del directorio donde se encuentra, un puntero a una estructura con los atributos nuevos del fichero (si no se quiere utilizar póngase a NULL), se rellena en caso de éxito y el manejador del directorio obtenido en caso de éxito. Crea un cliente RPC para el *mount*, *mi_nfs_clnt*, por medio de la función *mi_nfs_create()* y realiza la petición al servidor, *rpc_call().* Recibe el manejador del directorio y los atributos del fichero en la variable *res* y el estado, cero en caso de éxito, devueltos por el servidor y los devuelve como parámetros de salida.

```
	int mi_lookup(struct sockaddr_in *sin, char *dir, int
	proto,  int  puerto,  struct  fhstatus  *fhin,  struct
	nfs_fattr *fattr, struct fhstatus *fhout)
	{
		struct rpc_clnt	*mi_nfs_clnt;
		char		hostname[FHSIZE];
		diropargs	arg;
		diropres	res;

		strcpy(hostname, in_ntoa(sin->sin_addr.s_addr));
		if  (!(mi_nfs_clnt  =  mi_nfs_create(hostname,  sin,
		proto, puerto)))
```

```
			return -EACCES;

		memcpy(arg.dir,fhin->fhstatus_u.directory, FHSIZE);
		arg.name = dir;
		fattr->valid = 0;

		if((rpc_call(mi_nfs_clnt,   NFSPROC_LOOKUP,   &arg,
		&res, 0)) == NFS_OK){
			memset(fhout->fhstatus_u.directory,         0,
			sizeof(fhandle));
			memcpy(fhout->fhstatus_u.directory,
			res.diropres_u.fhand_attr.file, FHSIZE);
			fhout->status                               =
			res.diropres_u.fhand_attr.attributes.type;

			if(fattr != NULL){
				memcpy(fattr,
				&res.diropres_u.fhand_attr.attributes,
				sizeof(fattr));
				printk("mi_nfs>       NFS>         RPC>
				Atributos:\n");
				//  Mensajes  por  pantalla  con  los
				atributos  del  objeto  del  sistema  de
				ficheros
			}
		}
		else
			fhout->status = res.status;
		return res.status;
	}
```

7.3.2.6 Readlink

Esta función lee los enlaces simbólicos o blandos. Recibe como parámetros de entrada el socket, el protocolo, el puerto, el manejador del enlace y el path obtenido en caso de éxito. Crea un cliente RPC para el *mount*, *mi_nfs_clnt*, por medio de la función *mi_nfs_create()* y realiza la petición al servidor, *rpc_call().* Recibe el *path* y el estado, cero en caso de éxito, devueltos por el servidor y los devuelve como parámetros de salida.

```
	int mi_readlink(struct sockaddr_in *sin, int proto, int
	puerto, fhandle *fh, char *path)
	{
		struct rpc_clnt	*mi_nfs_clnt;
		char		hostname[FHSIZE];
		readlinkres	clnt_res;
		int		len = 0;

		strcpy(hostname, in_ntoa(sin->sin_addr.s_addr));
		if  (!(mi_nfs_clnt  =  mi_nfs_create(hostname,  sin,
		proto, puerto)))
			return -EACCES;
```

```
		if (clnt_res.readlinkres_u.data == NULL)
			return -EACCES;

		if(rpc_call(mi_nfs_clnt,	NFSPROC_READLINK,	fh,
		&clnt_res, 0) == NFS_OK){
			len = strlen (clnt_res.readlinkres_u.data);
			memset(path, 0, sizeof(path));
			strncpy(path	,	clnt_res.readlinkres_u.data,
			len);
		}
		return clnt_res.status;
	}
```

7.3.2.7 Read

Esta función lee datos de un fichero. Recibe como parámetros de entrada el socket, el protocolo, el puerto, el manejador del fichero del que vamos a leer, el offset o desplazamiento del fichero, que indica desde donde empezar a leer, el buffer que contendrá los datos leídos en caso de éxito, el tamaño de los datos que se desean leer y el tamaño de bloque del fichero. *Args* es el argumento de entrada para la llamada RPC, *res* es el argumento de salida e *i* es el contador, indica cuantos bytes se han leído, inicialmente es igual cero. Mediante un bucle se leen los datos del fichero hasta que se lean todos los solicitados. Dentro del bucle se determina el tamaño de datos a leer en la llamada RPC y se introduce en la variable de entrada de la petición, así como el manejador del fichero a leer, el offset, el cual se incrementa con los ya datos leídos en las anteriores llamada RPC y el buffer, situándose en la siguiente posición a la última leída. Se crea un cliente RPC para el mount, *mi_nfs_clnt*, por medio de la función *mi_nfs_create()* y realiza la petición al servidor, *rpc_call().* Recibe los datos leídos y la longitud de los mismos en la variable *res* y el estado, cero en caso de éxito, devueltos por el servidor, se incrementa el tamaño leído, se copian los datos al final del buffer y por último devuelve el estado y el buffer con los datos leídos.

```
	int mi_read(struct sockaddr_in *sin, int proto, int
	puerto, fhandle *fh, long offset, char *buffer, long
	tam, int blockSize)
	{
		struct rpc_clnt	*mi_nfs_clnt;
		char		hostname[FHSIZE];
		int		estado = 0;
		readargs	args;
		readres		res;
		int		i = 0, j = 0, len = 0;

		if(tam <= 0)
			return NFS_OK;

		if(blockSize == 0)
			return -7;

		if((blockSize > MAXDATA)||(blockSize <= 0))
			blockSize = MAXDATA;
```

```
            while(i < tam){
                  if((tam - i) >= MAXREAD)
                        len = MAXREAD;
                  else
                        len = tam - i;

                  memcpy(args.file, fh, FHSIZE);
                  args.offset = offset + i;
                  args.count = len;
                  res.readres_u.fich_read.data.nfsdata_val     =
                  buffer + i;

                  strcpy(hostname,                    in_ntoa(sin-
                  >sin_addr.s_addr));
                  if  (!(mi_nfs_clnt  =  mi_nfs_create(hostname,
                  sin, proto, puerto)))
                        return -EACCES;

                  if((estado  =  rpc_call(mi_nfs_clnt,  nfs_read,
                  &args, &res, 0)) == NFS_OK){
                        j                                     =
                        res.readres_u.fich_read.data.nfsdata_len
                        ;
                        i = j + i;
                        strncat(buffer,
                        res.readres_u.fich_read.data.nfsdata_val
                        , j);

                        if (j < len)
                              return res.status;
                  }
                  else
                        return res.status;
            }
            return res.status;
      }
```

7.3.2.8 Write

Esta función escribe datos en un fichero. Recibe como parámetros de entrada el socket, el protocolo, el puerto, el manejador del fichero en el que vamos a escribir, el offset o desplazamiento del fichero, que indica desde donde empezar a escribir, el buffer con los datos que se desean escribir, el tamaño de los datos que se desean escribir, el tamaño de bloque del fichero y un puntero a una estructura con los atributos del fichero (si no se quiere utilizar póngase a NULL), que en caso de éxito se rellenará. *Args* es el argumento de entrada para la llamada RPC, *res* es el argumento de salida e *i* es el contador, indica cuantos bytes se han escrito, inicialmente es igual cero. Mediante un bucle se escriben los datos en el fichero hasta que se escriban todos los solicitados. Dentro del bucle se determina el tamaño de datos a escribir en la llamada RPC y se introduce en la variable de entrada de la petición, así como el manejador del fichero en el que vamos a escribir, el offset, el cual se incrementa con los datos

ya escritos en anteriores llamadas RPC y el buffer, situándose en la siguiente posición a escribir. Se crea un cliente RPC para el mount, *mi_nfs_clnt*, por medio de la función *mi_nfs_create()* y realiza la petición al servidor, *rpc_call()*. Recibe los atributos del fichero en la variable *res* y el estado, cero en caso de éxito, devueltos por el servidor, se incrementa el tamaño escrito y devuelve el estado.

```
int mi_write(struct sockaddr_in *sin, int proto, int
puerto, fhandle *fh, char *data, long offset, long size,
int blockSize, struct nfs_fattr *res)
{
      struct rpc_clnt *mi_nfs_clnt;
      char            hostname[FHSIZE];
      writeargs       args;
      int             estado = 0;
      int             i = 0, len = 0;

      if(blockSize == 0)
            return -7;

      if(size <= 0)
            return NFS_OK;

      if((blockSize > MAXDATA)||(blockSize <= 0))
            blockSize = MAXDATA;

      res->valid = 0;

      while(i < size){
            if((size - i) >= MAXDATA)
                  len = MAXDATA;
            else
                  len = size - i;

            memcpy(args.file, fh, FHSIZE);
            args.offset = offset + i;
            args.totalcount = size;
            args.data.nfsdata_val = data + i;
            args.data.nfsdata_len = len;

            strcpy(hostname,                  in_ntoa(sin-
            >sin_addr.s_addr));
            if  (!(mi_nfs_clnt  =  mi_nfs_create(hostname,
            sin, proto, puerto)))
                  return -EACCES;

            if((estado = rpc_call(mi_nfs_clnt, nfs_write,
            &args, res, 0)) != NFS_OK)
                  return estado;

            i = i + len;
      }
```

```
        if(res != NULL){
            printk("mi_nfs> NFS> RPC> Atributos:\n");
            // Mensajes por pantalla con los atributos
            del objeto del sistema de ficheros
        }
        return NFS_OK;
    }
```

7.3.2.9 Create

Función para crear un fichero. Recibe como parámetros de entrada el socket, el protocolo, el puerto, el nombre del fichero que se desea crear, el manejador del directorio donde se va a crear el fichero, los permisos de acceso del fichero a crear, un puntero a una estructura con los atributos del fichero (si no se quiere utilizar póngase a NULL), que en caso de éxito se rellenará y el manejador del fichero creado, que en caso de éxito se rellenará. *Arg* es el argumento de entrada para la llamada RPC y *res* es el argumento de salida. Se crea un cliente RPC para el mount, *mi_nfs_clnt*, por medio de la función *mi_nfs_create()*. Se introducen los datos en la estructura de entrada de la llamada RPC y realiza la petición al servidor, *rpc_call().* Recibe el manejador del fichero creado y atributos en la variable *res* y el estado, cero en caso de éxito, devueltos por el servidor. Devuelve el estado el manejador y los atributos del nuevo fichero.

```
    int mi_create(struct sockaddr_in *sin, char *file, int
    proto, int puerto, struct fhstatus *fhin, unsigned int
    permisos, struct nfs_fattr *fattr, struct fhstatus
    *fhout)
    {
        struct rpc_clnt  *mi_nfs_clnt;
        char             hostname[FHSIZE];
        createargs       arg;
        diropres         res;

        strcpy(hostname, in_ntoa(sin->sin_addr.s_addr));
        if (!(mi_nfs_clnt = mi_nfs_create(hostname, sin,
        proto, puerto)))
            return -EACCES;

        memcpy(arg.where.dir,   fhin->fhstatus_u.directory,
        FHSIZE);
        arg.where.name = file;
        setAttr(&arg.attributes, permisos);
        fattr->valid = 0;

        if((res.status = rpc_call(mi_nfs_clnt, nfs_create,
        &arg, &res, 0)) != NFS_OK)
            return res.status;

        if(fhout != NULL)
            memcpy          (fhout->fhstatus_u.directory,
            res.diropres_u.fhand_attr.file, FHSIZE);

        if(fattr != NULL){
```

```
                memcpy(fattr,
                &res.diropres_u.fhand_attr.attributes,
                sizeof(fattr));
                printk("mi_nfs> NFS> RPC> Atributos:\n");
                // Mensajes por pantalla con los atributos
                del objeto del sistema de ficheros
        }
        return NFS_OK;
}
```

7.3.2.10 Remove

Función para borrar un fichero. Recibe como parámetros de entrada el socket, el nombre del fichero que se desea borrar, el protocolo, el puerto y el manejador del directorio donde se encuentra el fichero a borrar. *Args* es el argumento de entrada para la llamada RPC. Se crea un cliente RPC para el *mount*, *mi_nfs_clnt*, por medio de la función *mi_nfs_create()*. Se introducen los datos en la estructura de entrada de la llamada RPC y realiza la petición al servidor, *rpc_call()*. Recibe el estado, cero en caso de éxito, devuelto por el servidor y lo devuelve.

```
int mi_remove(struct sockaddr_in *sin, char *file, int
proto, int puerto, fhandle fh)
{
        struct rpc_clnt *mi_nfs_clnt;
        char            hostname[FHSIZE];
        int             estado = 0;
        diropargs       args;

        strcpy(hostname, in_ntoa(sin->sin_addr.s_addr));
        if (!(mi_nfs_clnt = mi_nfs_create(hostname, sin,
        proto, puerto)))
                return -EACCES;

        memcpy(args.dir, fh, FHSIZE);
        args.name = file;

        estado = rpc_call(mi_nfs_clnt, nfs_remove, &args,
        NULL, 0);
        return estado;
}
```

7.3.2.11 Rename

Esta función renombra un objeto del sistema de ficheros. Recibe como parámetros de entrada el socket, el protocolo, el puerto, el nombre del objeto que se desea renombrar, el manejador del directorio donde se encuentra el objeto a renombrar y el nuevo nombre del objeto. *Args* es el argumento de entrada para la llamada RPC. Se crea un cliente RPC para el mount, *mi_nfs_clnt*, por medio de la función *mi_nfs_create()*. Se introducen los datos en la estructura de entrada de la llamada RPC y realiza la petición al servidor, *rpc_call()*. Recibe el estado, cero en caso de éxito, devuelto por el servidor y lo devuelve.

```
int mi_rename(struct sockaddr_in *sin, int proto, int
puerto, char *name, fhandle fh, char *nameR)
```

```
{
	struct rpc_clnt	*mi_nfs_clnt;
	char		hostname[FHSIZE];
	int		estado = 0;
	renameargs	args;

	strcpy(hostname, in_ntoa(sin->sin_addr.s_addr));
	if (!(mi_nfs_clnt = mi_nfs_create(hostname, sin,
	proto, puerto)))
		return -EACCES;

	memcpy(args.from.dir, fh, FHSIZE);
	args.from.name = name;
	memcpy(args.to.dir, fh, FHSIZE);
	args.to.name = nameR;

	estado = rpc_call(mi_nfs_clnt, NFSPROC_RENAME,
	&args, NULL, 0);
	return estado;
}
```

7.3.2.12 Link

Esta función crea enlaces físicos o duros. Recibe como parámetros de entrada el socket, el protocolo, el puerto, el manejador del objeto al que referencia el enlace duro, el manejador del directorio donde se va a crear el enlace duro y el nombre del enlace duro. *Args* es el argumento de entrada para la llamada RPC. Se crea un cliente RPC para el mount, *mi_nfs_clnt*, por medio de la función *mi_nfs_create()*. Se introducen los datos en la estructura de entrada de la llamada RPC y realiza la petición al servidor, *rpc_call()*. Recibe el estado, cero en caso de éxito, devuelto por el servidor y lo devuelve.

```
int mi_link(struct sockaddr_in *sin, int proto, int
puerto, fhandle FhPath, fhandle FhDir, char *name)
{
	struct rpc_clnt	*mi_nfs_clnt;
	char		hostname[FHSIZE];
	int		estado = 0;
	linkargs	args;

	strcpy(hostname, in_ntoa(sin->sin_addr.s_addr));
	if (!(mi_nfs_clnt = mi_nfs_create(hostname, sin,
	proto, puerto)))
		return -EACCES;

	memcpy(args.from, FhPath, FHSIZE);
	memcpy(args.to.dir, FhDir, FHSIZE);
	args.to.name = name;

	estado = rpc_call(mi_nfs_clnt, nfs_link, &args,
	NULL, 0);
	return estado;
}
```

7.3.2.13 Symlink

Esta función crea enlaces simbólicos o blandos. Recibe como parámetros de entrada el socket, el protocolo, el puerto, el manejador del directorio donde se va a encontrar el enlace simbólico, el nombre del enlace, el path al que apunta el enlace simbólico y los permisos que tendrá el enlace. *Args* es el argumento de entrada para la llamada RPC. Se crea un cliente RPC para el mount, *mi_nfs_clnt*, por medio de la función *mi_nfs_create()*. Se introducen los datos en la estructura de entrada de la llamada RPC y realiza la petición al servidor, *rpc_call()*. Recibe el estado, cero en caso de éxito, devuelto por el servidor y lo devuelve.

```
int mi_symlink(struct sockaddr_in *sin, int proto, int
puerto, fhandle fh, char *name, char *path, unsigned int
permisos)
{
      struct rpc_clnt *mi_nfs_clnt;
      char            hostname[FHSIZE];
      int             estado = 0;
      symlinkargs     args;

      strcpy(hostname, in_ntoa(sin->sin_addr.s_addr));
      if  (!(mi_nfs_clnt  =  mi_nfs_create(hostname,  sin,
      proto, puerto)))
            return -EACCES;

      memcpy(args.from.dir, fh, FHSIZE);
      args.from.name = name;
      args.to = path;
      setAttr(&args.attributes, permisos);

      estado  = rpc_call(mi_nfs_clnt, nfs_symlink, &args,
      NULL, 0);
      return estado;
}
```

7.3.2.14 Mkdir

Función para crear un directorio. Recibe como parámetros de entrada el socket, el protocolo, el puerto, el manejador del directorio donde se va a crear el nuevo directorio, el nombre del directorio que se desea crear, los permisos de acceso del directorio a crear, el manejador del directorio creado, se rellenará en caso de éxito y un puntero a una estructura con los atributos del directorio (si no se quiere utilizar póngase a NULL), que en caso de éxito se rellenara. *Args* es el argumento de entrada para la llamada RPC y *res* el argumento de salida. Se crea un cliente RPC para el mount, *mi_nfs_clnt*, por medio de la función *mi_nfs_create()*. Se introducen los datos en la estructura de entrada de la llamada RPC y realiza la petición al servidor, *rpc_call()*. Recibe en la estructura *res* el manejador del nuevo directorio creado y el estado, cero en caso de éxito, devueltos por el servidor y los devuelve como parámetros de salida.

```
int mi_mkdir (struct sockaddr_in *sin, int proto, int
puerto, fhandle fhin, char *dir, unsigned int mode,
fhandle fhout, struct nfs_fattr *fattr)
{
	struct rpc_clnt	*mi_nfs_clnt;
	char		hostname[FHSIZE];
	createargs	args;
	diropres	res;

	strcpy(hostname, in_ntoa(sin->sin_addr.s_addr));
	if (!(mi_nfs_clnt = mi_nfs_create(hostname, sin,
	proto, puerto)))
		return -EACCES;

	memcpy(args.where.dir, fhin, FHSIZE);
	args.where.name = dir;
	setAttr(&(args.attributes), mode);
	fattr->valid = 0;

	if((res.status = rpc_call(mi_nfs_clnt, nfs_mkdir,
	&args, &res, 0)) == NFS_OK){
		if(fhout != NULL)
			memcpy			(fhout,
			res.diropres_u.fhand_attr.file, FHSIZE);

		if(fattr != NULL){
			memcpy(fattr,
			&(res.diropres_u.fhand_attr.attributes),
			sizeof(fattr));
			printk("mi_nfs>		NFS>		RPC>
			Atributos:\n");
			// Mensajes por pantalla con los
			atributos del objeto del sistema de
			ficheros
		}
	}
	return res.status;
}
```

7.3.2.15 Rmdir

Función para borrar un directorio. Recibe como parámetros de entrada el socket, el protocolo, el puerto, el manejador del directorio donde se encuentra el directorio a borrar y el nombre del directorio que se desea borrar. *Args* es el argumento de entrada para la llamada RPC. Se crea un cliente RPC para el mount, *mi_nfs_clnt*, por medio de la función *mi_nfs_create()*. Se introducen los datos en la estructura de entrada de la llamada RPC y realiza la petición al servidor, *rpc_call()*. Recibe el estado, cero en caso de éxito, devuelto por el servidor y lo devuelve como parámetros de salida.

```
int mi_rmdir(struct sockaddr_in *sin, int proto, int
puerto, fhandle fh, char *dir)
{
	struct rpc_clnt	*mi_nfs_clnt;
	char		hostname[FHSIZE];
	int		estado = 0;
	diropargs	args;

	strcpy(hostname, in_ntoa(sin->sin_addr.s_addr));
	if (!(mi_nfs_clnt = mi_nfs_create(hostname, sin,
	proto, puerto)))
		return -EACCES;

	memcpy(args.dir, fh, FHSIZE);
	args.name = dir;

	estado = rpc_call(mi_nfs_clnt, nfs_rmdir, &args,
	NULL, 0);
	return estado;
}
```

7.3.2.16 Statfs

Esta función obtiene las características del servidor NFS. Recibe como parámetros de entrada el socket, el protocolo, el puerto y el manejador del directorio. *Fh* es el argumento de entrada de la llamada RPC y *res* el argumento de salida. Se crea un cliente RPC para el mount, *mi_nfs_clnt*, por medio de la función *mi_nfs_create()*. Se introducen los datos en la estructura de entrada de la llamada RPC y realiza la petición al servidor, *rpc_call()*. Recibe el estado, cero en caso de éxito, devuelto por el servidor y lo devuelve como parámetros de salida.

```
int mi_statfs(struct sockaddr_in *sin, int proto, int
puerto, fhandle fh)
{
	struct rpc_clnt		*mi_nfs_clnt;
	char			hostname[FHSIZE];
	struct nfs_fsinfo	res;
	int			estado = 0;

	strcpy(hostname, in_ntoa(sin->sin_addr.s_addr));
	if (!(mi_nfs_clnt = mi_nfs_create(hostname, sin,
	proto, puerto)))
		return -EACCES;

	if((estado = rpc_call(mi_nfs_clnt, nfs_statfs, fh,
	&res, 0)) == NFS_OK){
		printk("mi_nfs> NFS> RPC> Estadísticas del
		sistema de ficheros:\n");
		// Mensajes por pantalla con las estadísticas
		del sistema de ficheros
	}
	return estado;
}
```

7.4 Funciones XDR

A continuación se muestran las cabeceras de las funciones de codificación y decodificación utilizadas y se explica su cometido.

7.4.1 Tipos de datos básicos

7.4.1.1 void

```
static int nfs_xdr_enc_void(struct rpc_rqst *req, u32
*p, void *dummy)
```

Pone en formato de red un argumento de tipo *void*.

```
static int nfs_xdr_dec_void(struct rpc_rqst *req, u32
*p, void *clnt_res)
```

Pone en formato máquina un argumento de tipo void.

7.4.1.2 string

```
static int xdr_encode_dirpath (struct rpc_rqst *req, u32
*p, char *objp)
```

Pone en formato de red un argumento de tipo *string*.

```
static inline u32 * mi_xdr_decode_string(u32 *p, char
**string, unsigned int *len, unsigned int maxlen)
```

Pone en formato máquina un argumento de tipo *string*.

7.4.1.3 fhandle

```
static inline u32 * xdr_encode_fhandle(u32 *p, fhandle
*fh)
```

Pone en formato de red un argumento de tipo *fhandle*.

```
static inline u32 * xdr_decode_fhandle(u32 *p, fhandle
*fhandle)
```

Pone en formato máquina un argumento de tipo *fhandle*.

7.4.1.4 fhstatus

```
static int xdr_decode_fhstatus(struct rpc_rqst *req, u32
*p, struct fhstatus *objp)
```

Pone en formato máquina un argumento de tipo *fhstatus*.

7.4.1.5 time

```
static inline u32* xdr_decode_time(u32 *p, u64 *timep)
```

Pone en formato máquina un argumento de tipo fecha.

7.4.1.6 nfs_fattr

```
static inline u32 * xdr_decode_fattr(u32 *p, struct
nfs_fattr *fattr)
```

Pone en formato máquina los valores de la estructura *nfs_fattr*.

7.4.1.7 iattr

```
static inline u32 * xdr_encode_sattr(u32 *p, struct
iattr *attr)
```

Pone en formato de red los valores de la estructura *iattr*.

7.4.2 Protocolo mount

```
static int nfs_xdr_enc_void(struct rpc_rqst *req, u32
*p, void *dummy)
```

Pone en formato de red un argumento de tipo *void*. Ya se he comentado en tipos de datos básicos. Es utilizado por ***nulo***, ***dump***, ***export*** y ***umountall***.

```
static int nfs_xdr_dec_void(struct rpc_rqst *req, u32
*p, void *clnt_res)
```

Pone en formato máquina un argumento de tipo *void*. Ya se he comentado en tipos de datos básicos. Es utilizado por **nulo**, **umount** y **umountall**.

```
static int xdr_encode_dirpath (struct rpc_rqst *req, u32
*p, char *objp)
```

Pone en formato de red un argumento de tipo *string*. Ya se he comentado en tipos de datos básicos. Es utilizado por ***mount*** y ***umount***.

```
static int xdr_decode_fhstatus(struct rpc_rqst *req, u32
*p, struct fhstatus *objp)
```

Pone en formato máquina un argumento de tipo *fhstatus*. Ya se he comentado en tipos de datos básicos. Es utilizado por ***mount***.

```
static int xdr_decode_mountlist(struct rpc_rqst *req,
u32 *p, struct mountbody *res)
```

Es una función recursiva. Pone en formato máquina la respuesta recibida de la petición ***dump***, el nombre de la máquina y el directorio.

```
static int xdr_decode_exports(struct rpc_rqst *req, u32
*p, struct exportnode *res)
```

Es una función recursiva. Pone en formato máquina la respuesta recibida de la petición ***export***, el path y el nombre de la máquina.

7.4.3 Protocolo NFS

```
static int nfs_xdr_enc_void(struct rpc_rqst *req, u32
*p, void *dummy)
```

Pone en formato de red un argumento de tipo *void*. Ya se he comentado en tipos de datos básicos. Es utilizado por ***null***, ***root*** y ***writecache***.

```
static int nfs_xdr_dec_void(struct rpc_rqst *req, u32
*p, void *clnt_res)
```

Pone en formato máquina un argumento de tipo *void*. Ya se he comentado en tipos de datos básicos. Es utilizado por ***null***, ***root*** y ***writecache***.

```
static int nfs_xdr_fhandle(struct rpc_rqst *req, u32 *p,
fhandle *fh)
```

Pone en formato de red un argumento de tipo fhandle, utiliza la función *xdr_encode_fhandle* comentada en tipos de datos básicos. Es utilizado por ***getattr***, ***readlink*** y ***statfs***.

```
static int nfs_xdr_attrstat(struct rpc_rqst *req, u32
*p, struct nfs_fattr *fattr)
```

Introduce en la estructura *nfs_fattr* los valores devueltos por el servidor en formato máquina. Es utilizado por ***getattr***, ***setattr*** y ***write***.

```
static int nfs_xdr_sattrargs(struct rpc_rqst *req, u32
*p, struct sattrargs *args)
```

Pone en formato de red los datos de la estructura *sattrargs*, utiliza las funciones xdr_encode_fhandle y xdr_encode_sattr comentadas en tipos de datos básicos. Es utilizado por ***setattr***.

```
static int nfs_xdr_diropargs(struct rpc_rqst *req, u32
*p, struct diropargs *args)
```

Pone en formato de red los datos de la estructura *diropargs*, utiliza las funciones *xdr_encode_fhandle* y *xdr_encode_string* comentadas en tipos de datos básicos. Es utilizado por ***lookup***, ***remove*** y ***rmdir***.

```
static int nfs_xdr_diropres(struct rpc_rqst *req, u32
*p, struct diropres *res)
```

Introduce en la estructura *diropres* los valores devueltos por el servidor en formato máquina. Utiliza la función *xdr_decode_fhandle* y *xdr_decode_fattr* comentadas en tipos de datos básicos. Es utilizado por ***lookup***, ***create*** y ***mkdir***.

```
static int nfs_xdr_readlinkres(struct rpc_rqst *req, u32
*p, readlinkres *objp)
```

Introduce en la estructura *readlinkres* los valores devueltos por el servidor en formato máquina. Utiliza la función *mi_xdr_decode_string* comentada en tipos de datos básicos. Es utilizado por ***readlink***.

```
static int nfs_xdr_readargs(struct rpc_rqst *req, u32 *p,
readargs *objp)
```

Pone en formato de red los datos de la estructura *readargs*, utiliza las funciones *xdr_encode_fhandle* comentada en tipos de datos básicos. Es utilizado por ***read***.

```
static int nfs_xdr_readres(struct rpc_rqst *req, u32 *p,
readres *objp)
```

Introduce en la estructura *readres* los valores devueltos por el servidor en formato máquina. Utiliza la función *xdr_decode_fattr* y *xdr_decode_string* comentadas en tipos de datos básicos. Es utilizado por ***read***.

```
static int nfs_xdr_writeargs(struct rpc_rqst *req, u32 *p,
writeargs *args)
```

Pone en formato de red los datos de la estructura *writeargs*, utiliza las funciones *xdr_encode_fhandle* comentada en tipos de datos básicos y *xdr_encode_array* definida en el kernel de Linux. Es utilizado por ***write***.

```
static int nfs_xdr_createargs(struct rpc_rqst *req, u32
*p, createargs *args)
```

Pone en formato de red los datos de la estructura *createargs*, utiliza las funciones *xdr_encode_fhandle, xdr_encode_string* y *xdr_encode_sattr* comentadas en tipos de datos básicos. Es utilizado por ***create***.

```
static int nfs_xdr_stat(struct rpc_rqst *req, u32 *p, void
*dummy)
```

Devuelve el estado devuelto por el servidor en formato máquina. Es utilizado por ***remove***, ***rename, link, symlink*** y ***rmdir***.

```
static int nfs_xdr_renameargs(struct rpc_rqst *req, u32
*p, renameargs *args)
```

Pone en formato de red los datos de la estructura *renameargs*, utiliza las funciones *xdr_encode_fhandle* y *xdr_encode_string* comentadas en tipos de datos básicos. Es utilizado por ***rename***.

```
static int nfs_xdr_linkargs(struct rpc_rqst *req, u32 *p,
linkargs *args)
```

Pone en formato de red los datos de la estructura *linkargs*, utiliza las funciones *xdr_encode_fhandle* y *xdr_encode_string* comentadas en tipos de datos básicos. Es utilizado por ***link***.

```
static int nfs_xdr_symlinkargs(struct rpc_rqst *req, u32
*p, symlinkargs *args)
```

Pone en formato de red los datos de la estructura *symlinkargs*, utiliza las funciones *xdr_encode_fhandle, xdr_encode_string* y *xdr_encode_sattr* comentadas en tipos de datos básicos. Es utilizado por ***symlink***.

```
static int nfs_xdr_statfsres(struct rpc_rqst *req, u32 *p,
struct nfs_fsinfo *res)
```

Introduce en la estructura *nfs_fsinfo* los valores devueltos por el servidor en formato máquina. Es utilizado por ***statfs***.

8 Evaluación

Durante el siguiente capítulo se expone cómo se ha llevado a cabo la evaluación del módulo cargable cliente de NFS, NFS LKM. Se definirá el entorno donde se han realizado las pruebas, las pruebas seleccionadas para evaluar el NFS LKM, el propósito de dichas pruebas, los resultados obtenidos en las pruebas y finalmente se concluirá con un resumen final de los resultados obtenidos.

8.1 Entorno de las pruebas

Las pruebas se han realizado en una red doméstica utilizando un cliente, el módulo NFS LKM implementado, y un servidor NFS de Linux. Las características de los computadores se describen a continuación:

- CPU: Quadcore Intel(R) Xeon(R) E5405 @ 2.00GHz
- Memoria Ram: 4 GB
- Red: 50 Mbps (Fibra óptica)

8.2 Definición de las pruebas

Se han realizado diez ejecuciones de las distintas operaciones a evaluar, todas ellas para un cliente NFS contra un servidor NFS v2, obteniendo los valores medios para ser representados gráficamente. Se agruparan en los conjuntos siguientes:

- Creación y borrado de un número determinado de ficheros:
  - Comprendidos entre 400 y 9000.
- Lectura de un fichero:
  - Con tamaños comprendidos entre 1 MB y 128 MB.
  - Con tamaño de bloque comprendidos entre 1 KB y 8 KB.
- Lectura y escritura de un fichero:
  - Con tamaños comprendidos entre 128 KB y 16 MB.
  - Con tamaño de bloque comprendidos entre 1 KB y 8 KB.

Para la realización de las pruebas del módulo cliente de NFS se han implementado programas en lenguaje C que permiten invocar a las distintas operaciones definidas a evaluar, tomando el tiempo del sistema antes de comenzar y tras finalizar las mismas. Para llevar a cabo las del cliente NFS del sistema operativo se han implementado Shell Scripts para invocar a llamadas del sistema de ficheros, se toman tiempos antes y después de su realización.

8.3 Propósito de las pruebas

El propósito es comparar el tiempo que tardan en ejecutarse las pruebas anteriormente descritas utilizando por un lado el cliente NFS del sistema operativo, NFS SSOO, y por otro el módulo cliente NFS implementado, NFS LKM.

Se pretenden analizar diversas situaciones que hagan ver las virtudes y carencias del módulo cliente NFS, frente al cliente NFS del sistema operativo.

8.4 Resultados

En esta sección se muestran los resultados obtenidos tras la ejecución de las distintas pruebas descritas sobre el modulo cliente NFS cargable y comparándolo con el cliente NFS del propio sistema operativo. Posteriormente se analizaran y comentaran los resultados obtenidos.

Se han realizado gráficas de dispersión con líneas suavizadas y marcadores de los tiempos de ejecución en segundos.

8.4.1 Creación y borrado

A continuación se muestran los tiempos obtenidos en la creación y borrado de *"n"* ficheros, donde *"n"* tomo los siguientes valores: 400, 600, 800, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000 y 9000.

Como podemos observar los tiempos de las operaciones de creación y borrado entre el LKM NFS implementado y NFS son muy similares. En este caso el modulo implementado registra mejores tiempos en lo que a velocidad se refiere.

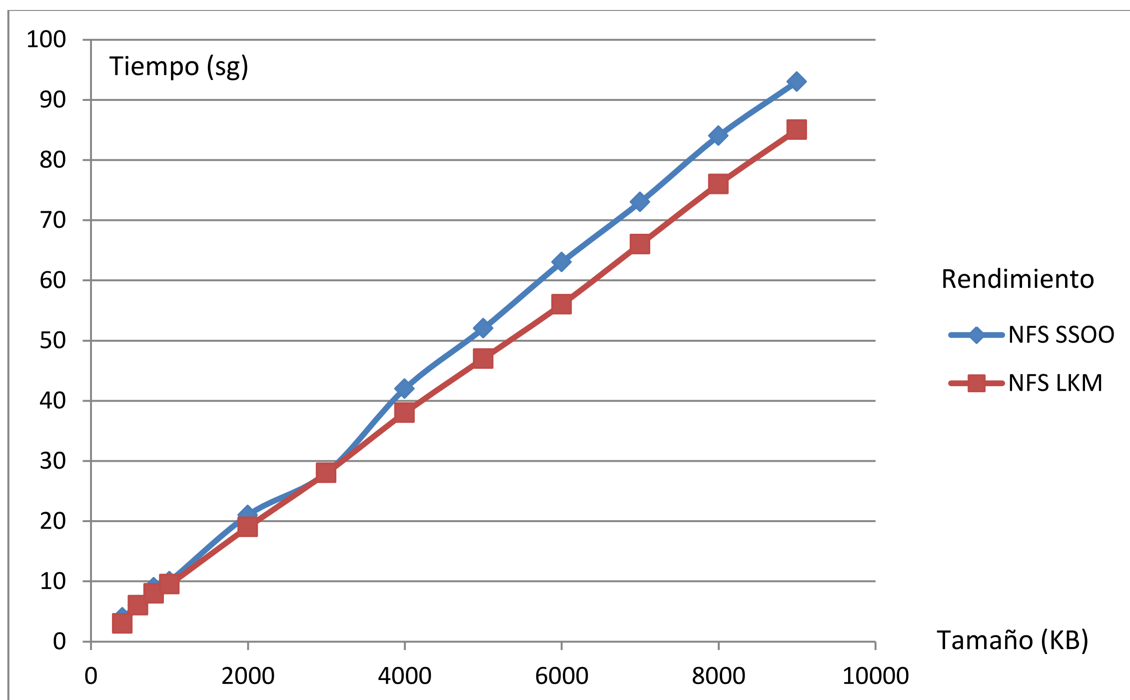

Figura 8.1 Creación y borrado de ficheros

8.4.2 Lectura

En el siguiente apartado se presentarán distintas gráficas en función del tamaño de bloque utilizado en la realización de las pruebas. Estas gráficas muestran los tiempos obtenidos en la lectura de un fichero de diferentes tamaños comprendidos entre 1 MB y 128 MB.

Como podemos observar los tiempos de la operación de lectura entre el LKM NFS implementado y NFS son muy similares, sufriendo pequeñas variaciones en función del tamaño de bloque empleado. Hasta que se incurre en un punto de inflexión en el que el tiempo del LKM NFS crece exponencialmente, si bien se ha comprobado esta misma tendencia en NFS a medida que se aumenta el tamaño del fichero.

Estas diferencias en el rendimiento están provocadas por el empleo de la caché de bloques en el cliente NFS del sistema operativo, no siendo así en el caso del LKM NFS implementado.

8.4.2.1 Tamaño de bloque 1 KB

En la realización de lecturas de ficheros con tamaño de bloque 1 KB, podemos observar como a partir de ficheros con tamaño mayor a 6 MB el rendimiento del módulo cargable NFS se empieza a resentir creciendo exponencialmente con respecto a NFS.

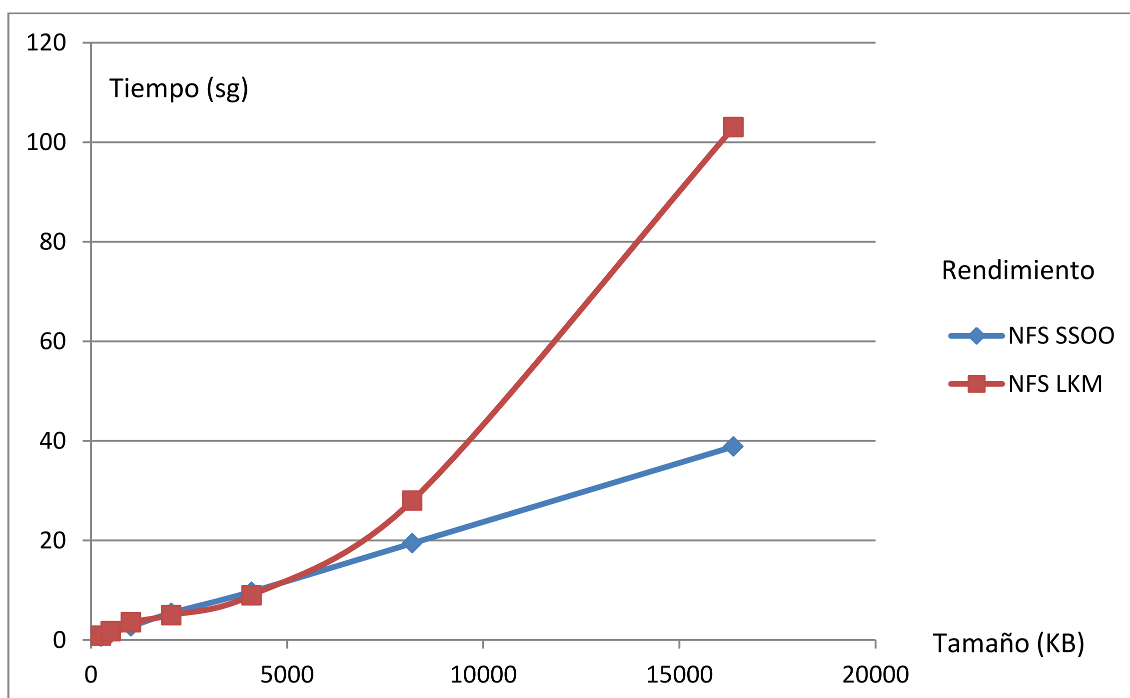

Figura 8.2 Lectura de ficheros en bloques de 1 KB

8.4.2.2 Tamaño de bloque 2 KB

En la realización de lecturas de ficheros con tamaño de bloque 2 KB, podemos observar como a partir de ficheros con tamaño mayor a 14 MB el rendimiento del módulo cargable NFS se empieza a resentir creciendo exponencialmente con respecto a NFS.

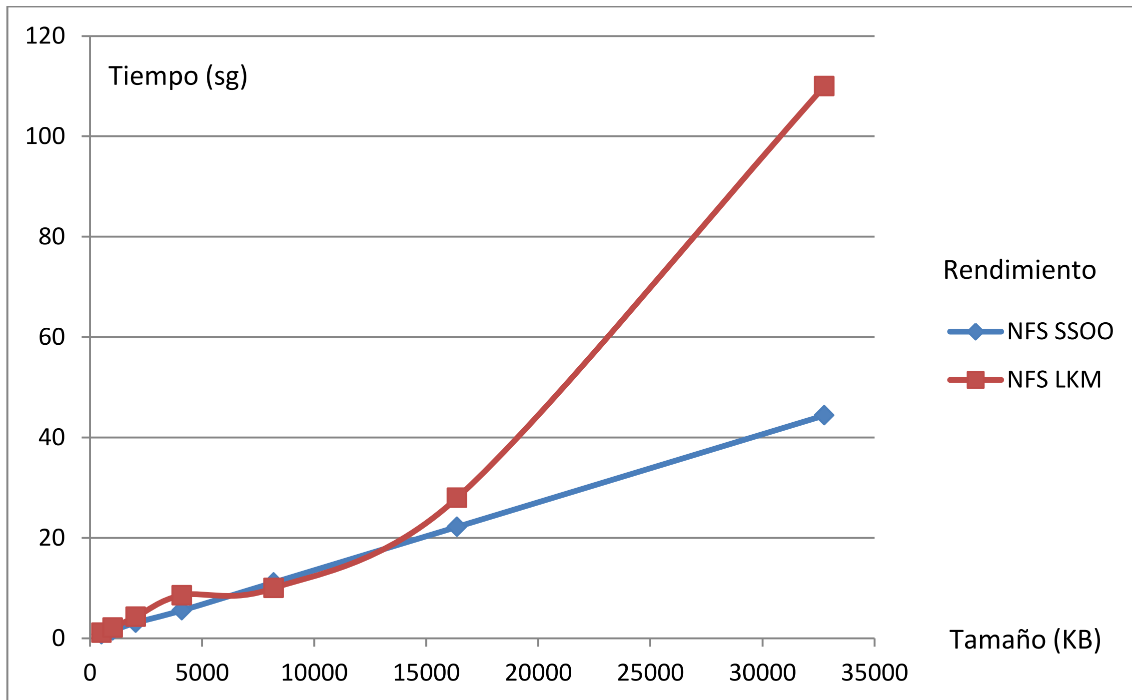

Figura 8.3 Lectura de ficheros en bloques de 2 KB

8.4.2.3Tamaño de bloque 4 KB

En la realización de lecturas de ficheros con tamaño de bloque 4 KB, podemos observar como a partir de ficheros con tamaño mayor a 30 MB el rendimiento del módulo cargable NFS se empieza a resentir creciendo exponencialmente con respecto a NFS. Además podemos ver cómo hasta dicho punto LKM mantiene un rendimiento ligeramente mejor.

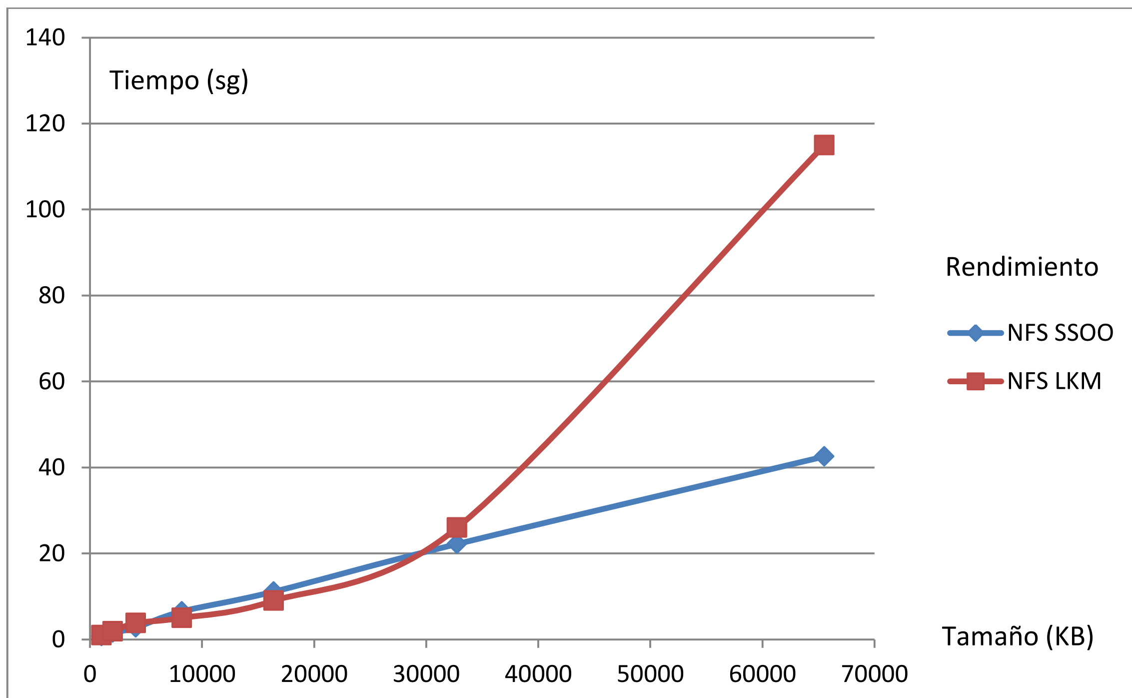

Figura 8.4 Lectura de ficheros en bloques de 4 KB

8.4.2.4Tamaño de bloque 8 KB

En la realización de lecturas de ficheros con tamaño de bloque 8 KB, podemos observar como a partir de ficheros con tamaño mayor a 70 MB el rendimiento del módulo cargable NFS se empieza a resentir creciendo exponencialmente con respecto a NFS.

Como indique al principio del apartado, en esta gráfica se observa claramente la tendencia de NFS a crecer exponencialmente el tiempo de respuesta a medida que los ficheros son mayores en cuanto a tamaño.

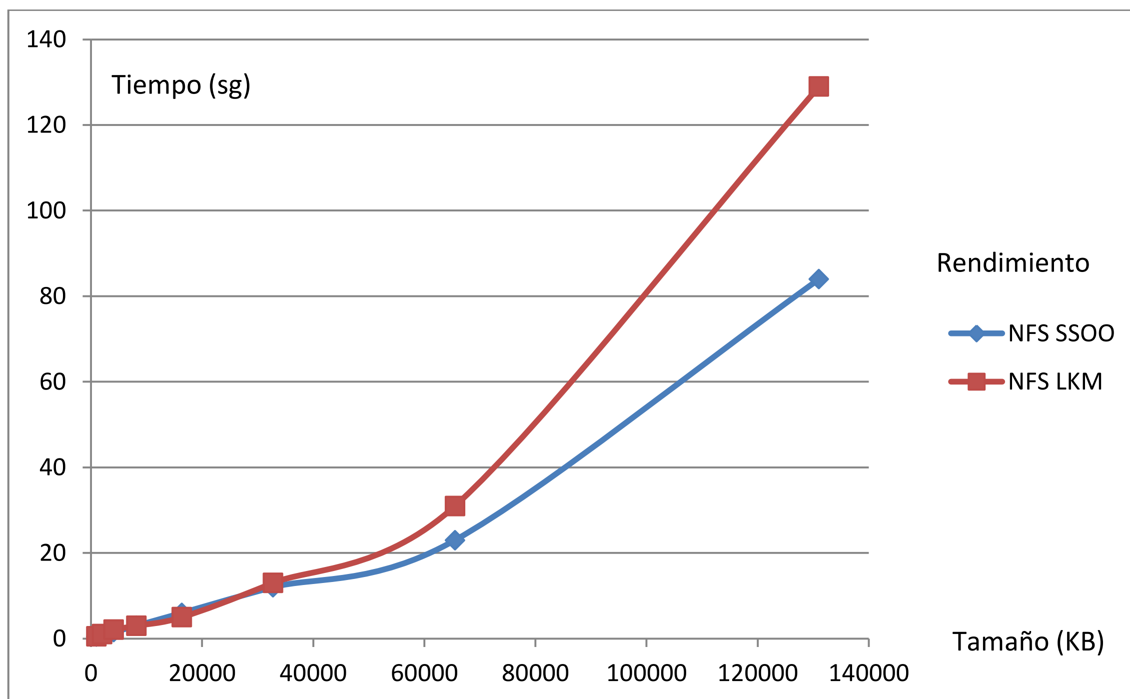

Figura 8.5 Lectura de ficheros en bloques de 8 KB

8.4.3 Lectura y escritura (copiar)

En esta sección se presentaran distintas gráficas en función del tamaño de bloque utilizado en la realización de las pruebas. Estas gráficas muestran los tiempos obtenidos en realizar una copia de un fichero de diferentes tamaños comprendidos entre 128 KB y 16 MB.

Como podemos observar los tiempos, de las operaciones necesarias para copiar un fichero, entre el LKM NFS implementado y NFS son muy similares. Sufriendo pequeñas variaciones en función del tamaño del fichero a copiar y del tamaño de bloque empleado.

8.4.3.1 Tamaño de bloque 1 KB

En la realización del copiado de ficheros con tamaño de bloque 1 KB, podemos observar un rendimiento prácticamente paralelo entre el modulo cargable NFS y NFS. Siendo mejor en tiempos, a mayor tamaño de fichero, el cliente del sistema de ficheros en red dotado por el sistema operativo.

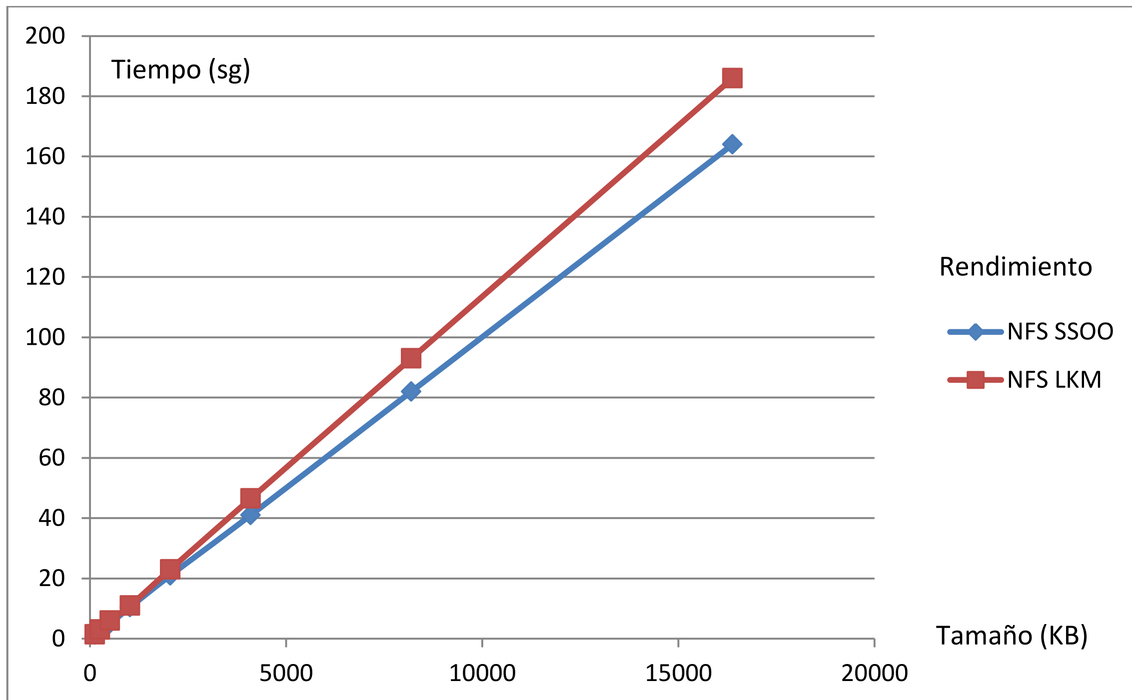

Figura 8.6 Lectura y Escritura de ficheros en bloques de 1 KB

8.4.3.2 Tamaño de bloque 2 KB

En la realización del copiado de ficheros con tamaño de bloque 2 KB, podemos observar un rendimiento prácticamente paralelo entre el modulo cargable NFS y NFS, siendo ligeramente mejor en tiempos el cliente dotado por el sistema operativo.

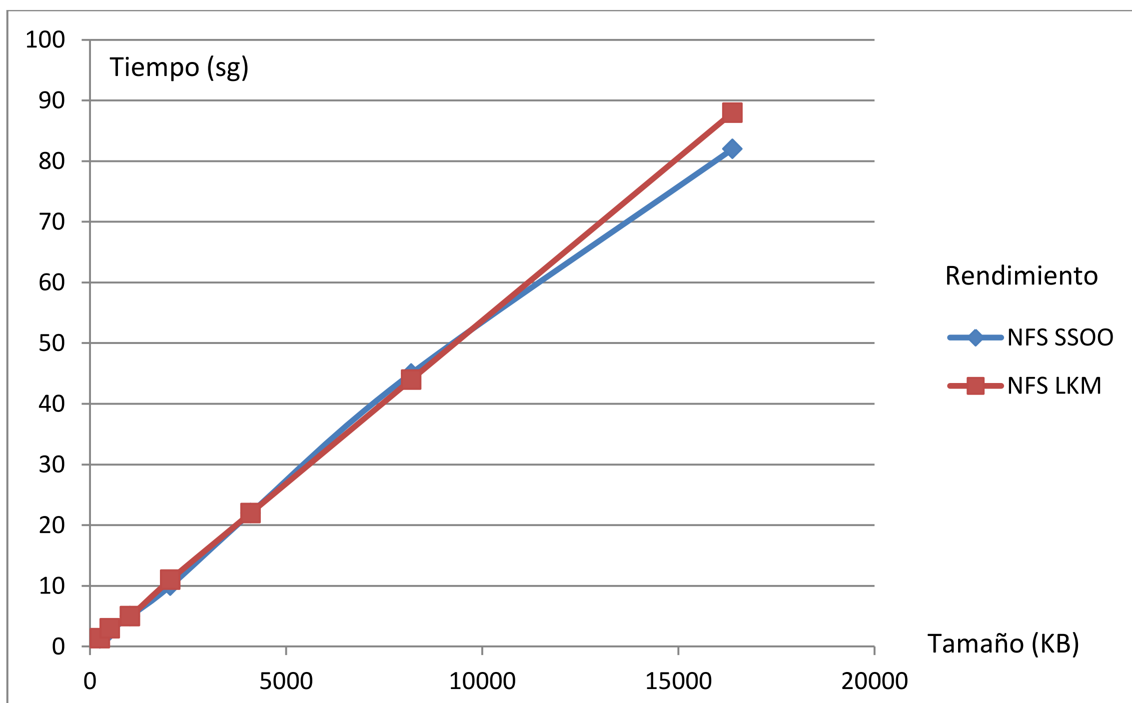

Figura 8.7 Lectura y Escritura de ficheros en bloques de 2 KB

8.4.3.3 Tamaño de bloque 4 KB

En el caso de copiado de ficheros con tamaño de bloque 4 KB, podemos observar como a partir de ficheros de 2 MB se produce en ligero distanciamiento en tiempos, que se mantienen prácticamente paralelos a medida que el tamaño del fichero a copiar aumenta.

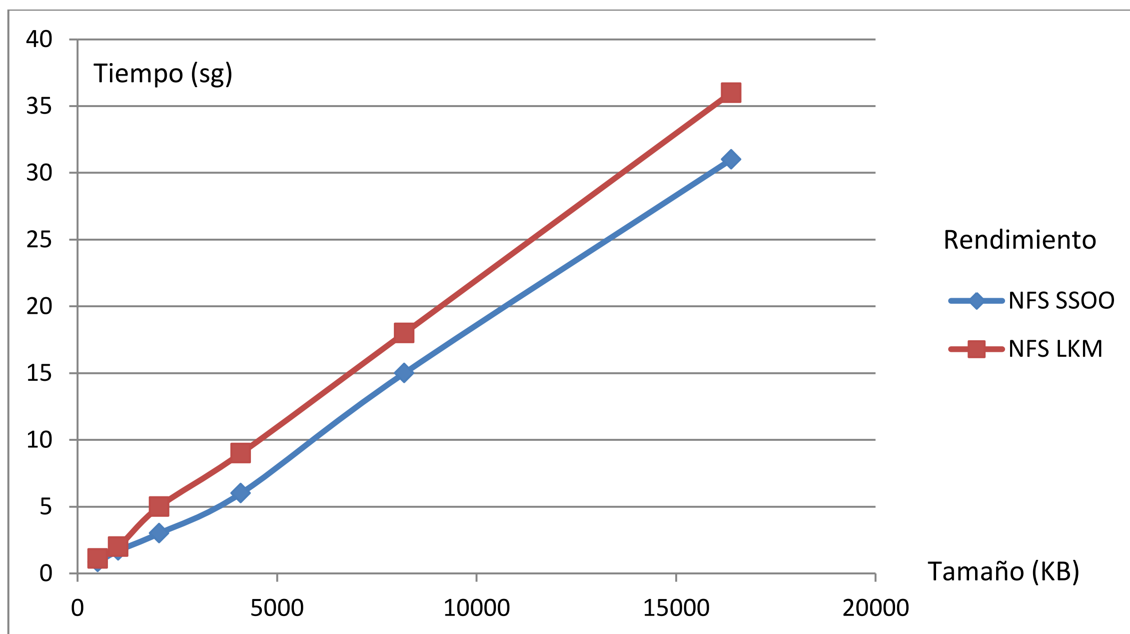

Figura 8.8 Lectura y Escritura de ficheros en bloques de 4 KB

8.4.3.4 Tamaño de bloque 8 KB

En el caso de copias de ficheros con tamaño de bloque de 8 KB, vemos que el rendimiento del LKM NFS es mejor hasta tamaños de ficheros de 3 MB, pasando a ser ligeramente peor en relación a NFS a partir de tamaños de fichero mayores.

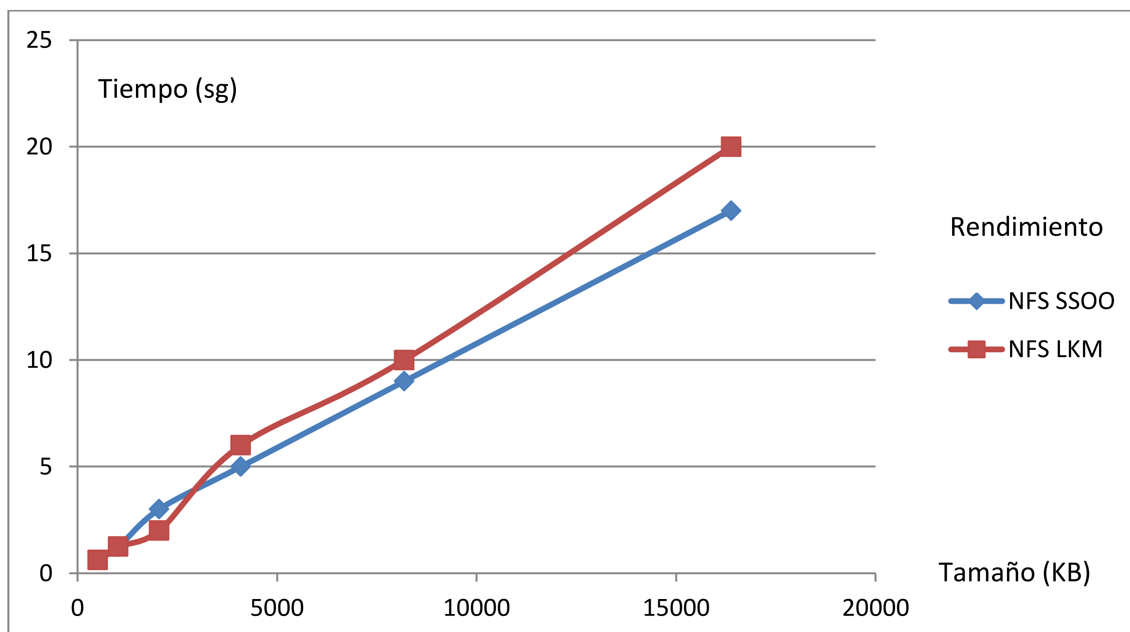

Figura 8.9 Lectura y Escritura de ficheros en bloques de 8 KB

8.5 Resumen

Tras las pruebas realizadas y analizadas anteriormente podemos decir que el módulo cargable presenta un óptimo rendimiento en relación a NFS. Si bien ha quedado patente que el tamaño ascendente de los ficheros a procesar, en las distintas operaciones y con los diferentes tamaños de bloque empleados, impacta exponencialmente en el rendimiento de este.

Es claramente observable en las gráficas relativas a la lectura de ficheros, en las que se observa un crecimiento exponencial del tiempo necesario para llevar a cabo la petición solicitada a medida que los tamaños de fichero aumentan.

La explicación de este hecho radica en la no utilización de caché de bloques en el módulo cargable que actúa como cliente NFS mientras que el cliente del propio sistema operativo si hace uso de la misma.

Como ya hemos comentado, NFS hace uso de tres tipos de caché en el cliente:

- Caché de nombres para acelerar las traducciones.
- Caché de metadatos del sistema de archivos: atributos de ficheros y directorios.
- Caché de bloques de ficheros y directorios.

Realiza *caching* mediante el uso de bloques de gran tamaño, típicamente 8 Kbyte, lo que proporciona lectura anticipada. La política de escritura es *write-through* retardada. Otra de las características es que sólo se envían al servidor bloques completos a excepción de cuando se cierra el fichero.

Periódicamente el cliente valida un bloque cargado en su caché comprobando en el servidor si los atributos han cambiado, mediante la operación *getattr* para obtener los atributos, y actualizando la copia en este caso. El periodo de validación suele ser de 3 segundos para ficheros y 30 segundos para directorios.

El empleo de caché de bloques permite mejorar el rendimiento:

- Explota el principio de proximidad de referencias:
    - Proximidad temporal.
    - Proximidad espacial.
- Lecturas adelantadas:
    - Mejora el rendimiento de las operaciones de lectura, sobre todo si son secuenciales
- Escrituras diferidas:
    - Mejora el rendimiento de las escrituras.

Si bien esta es una de las limitaciones que muestra NFS, el mantenimiento de la consistencia UNIX resulta problemático. La disminución del periodo de validación para mejorar la consistencia produce una sobrecarga por la gran cantidad de operaciones *getattr* que se realizan.

9 Presupuesto del proyecto

En el siguiente apartado se abordarán todos los aspectos concernientes al presupuesto final del proyecto.

9.1 Costes del personal

En la siguiente tabla se detalla el cálculo del coste por hora de los diferentes roles que han intervenido en la realización del proyecto.

	Jefe de proyecto		Analista		Programador	
Días laborables / mes	20 días					
Horas trabajadas / día	8 horas					
Horas trabajadas / mes	160 horas					
Salario neto / mes	2.800 €		2.200 €		1.600 €	
IRPF	35%	980 €	30%	660 €	25%	400 €
Seguridad Social	50%	1.400 €	45%	990 €	35%	560 €
Salario bruto / mes	5.180 €		3.850 €		2.560 €	
Salario bruto / año (14 pagas)	72.520 €		53.900 €		35.840 €	
Horas trabajadas / año	1760					
Coste bruto persona / hora	**41 €**		**31 €**		**20 €**	

Figura 9.1 Calculo coste persona/hora

A continuación se detallan las horas trabajadas por persona en cada fase del proyecto:

Actividad	Jefe de proyecto	Analista	Programador
Análisis	90 h	350 h	
Diseño	40 h	250 h	
Implementación	30 h		520 h
Pruebas		50 h	160 h
Documentación	20 h	150 h	100 h
Total	**180 h**	**800 h**	**780 h**

Figura 9.2 Horas trabajadas

En la siguiente tabla se describe el coste derivado del personal:

Perfil	Coste/hora	Horas trabajadas	Coste
Jefe de proyecto	41 €	180	7.380 €
Analista	31 €	800	24.800 €
Programador	20 €	780	15.600 €
		Total	**47.780 €**

Figura 9.3 Coste personal

9.2 Coste Hardware y licencias

En la siguiente tabla se desglosa el coste de hardware y licencias utilizados para el desarrollo del proyecto:

Descripción	Coste
Ordenador sobremesa	**510 €**
Procesador: Intel Dual Core E5400	170 €
Placa base: Asus P5KPL-AM IN Socket 775	150 €
Disco duro: 1TB SATA2	50 €
Memoria: Kingston 4Gb DDR2 800MHz	60 €
Tarjeta gráfica: Integrada	0 €
Tarjeta de sonido: Integrada	0 €
Fuente de alimentación: 500W	80 €
Ordenador portátil	**800 €**
Asus ul50vt	800 €
Licencias y software	**140 €**
Sistema operativo: Mandrake 8.1	0 €
Microsoft Office 2010 Hogar y estudiantes	140 €
Total	**1.450 €**

Figura 9.4 Coste hardware y licencias

9.3 Coste material fungible

Detalle del coste de material fungible:

Descripción	Coste
Material Oficina	**40 €**
Folios	25 €
Bolígrafos	15 €
Total	**40 €**

Figura 9.5 Coste material fungible

9.4 Coste oficina

Desglose del coste derivado de la oficina:

Descripción	Coste
Alquiler	**580 €**
Oficina	500 €
Acceso a internet	30 €
Luz	50 €
Mobiliario	400 €
Escritorio	150 €
Silla oficina	250 €
Total (12 meses)	**7.360 €**

Figura 9.6 Coste oficina

9.5 Coste final del proyecto

En la siguiente tabla se resume el coste final del proyecto:

Descripción	Coste
Coste de personal	47.780 €
Equipos informáticos y licencias	1.450 €
Material fungible	40 €
Oficina	7.360 €
Coste directos	56.630 €
Costes indirectos = 15% sobre Costes directos	8.495 €
Riesgo = 20% sobre Coste de personal	9.556 €
Total costes = Costes directos + Costes indirectos + Riesgo	74.681 €
Beneficio = 15% sobre Total costes	11.202 €
Total	**85.883 €**

Figura 9.7 Coste final del proyecto

10 Conclusiones

Se han cumplido los objetivos propuestos inicialmente en el apartado 1.1:

- Conocer el desarrollo de módulos del kernel:
  - Se ha realizado un estudio de la implementación de módulos cargables en Linux.
  - Se ha diseñado e implementado un cliente y servidor en espacio de usuario, calcular.
  - Se ha diseñado e implementado un módulo cliente cargable para calcular.
- Conocer el funcionamiento de NFS:
  - Se ha realizado un estudio del protocolo NFS y MOUNT y se ha integrado en la implementación del módulo cargable.
- Desarrollar un módulo del kernel de Linux que se comporte como cliente de NFS:
  - Se ha realizado un estudio de las llamadas a procedimientos remotos en el kernel y de su implementación en un módulo.
  - Se ha realizado un estudio del protocolo NFS y MOUNT.
  - Se ha diseñado e implementado un módulo cliente basado en las especificaciones dadas por el protocolo NFS y MOUNT.
- Permitir la comunicación entre procesos:
  - Se han implementado un conjunto de llamadas de RPC, *Remote Procedure Call*, y XDR, *eXternal Data Representation*, estas últimas son necesarias para poder realizar con éxito la comunicación entre máquinas que pueden ejecutar en arquitecturas diferentes.
- Realizar una evaluación del mismo:
  - Se ha analizado el rendimiento del módulo cargable en la realización de distintas operaciones comparando este con el de NFS.

Por tanto este proyecto pone en manifiesto cómo se puede diseñar e implementar cualquier módulo del kernel que actúe como cliente de un servidor, de cualquier tipo de servicios, en Linux por medio de llamadas a procedimientos remotos. También establece las bases para la creación de módulos servidores. Se puede decir que la memoria o documentación de este proyecto podría servir de guía o manual para el diseño e implementación de módulos cargables en el kernel de Linux. Así como de sistemas distribuidos con arquitectura cliente / servidor por medio de llamadas a procedimientos remotos.

10.1 Trabajos futuros

Para ampliar las posibilidades que puede ofrecer este Proyecto Fin de Carrera, se propone implementar las siguientes funcionalidades:

- Realización de una interfaz de usuario basada en las llamadas al sistema POSIX.
- Diseño e implementación de una caché para el modulo cliente NFS.
- Creación de un sistema de ficheros paralelo a partir del módulo cliente NFS.

11 Bibliografía

- Los Secretos del Kernel:
  http://www.linuxfocus.org/Castellano/January1998/article19.html

- Presentación de los servicios RPC:
  http://www.bulma.net/

- Laboratorio de Arquitectura de Ordenadores:
  http://www.gul.uc3m.es/gul/acortes/apuntes_lao/lao/Apuntes_de_L_A_O__Versión_0_3_3_.htm

- The Linux Kernel:
  http://www.tldp.org/LDP/tlk/tlk.html

- Linux Kernel/Núcleo
  http://www.linux-es.org/kernel

- Caso de Estudio: UNIX:
  http://sis-operativo.tripod.com/id10.html

- Linux Programación:
  http://www.paginadeprecada/linux-programacion.html

- Controladores de dispositivos de Linux, tercera edición:
  http://lwn.net/Kernel/LDD3/

- Building and Running Modules:
  http://static.lwn.net/images/pdf/LDD3/ch02.pdf

- Módulo de E/S:
  http://ldc.usb.ve/~spd/Docencia/ci-3821/Tema2/node5.html

- Dentro del núcleo Linux 2.4:
  http://ibiblio.org/pub/linux/docs/LuCaS/Manuales-LuCAS/DENTRO-NUCLEO-LINUX/dentro-nucleo-linux-html/dentro-nucleo-linux.html#toc2

- The File system:
  http://jcoppens.com/univ/data/pdf/fs/ext2fs_es.pdf

- TLDP-ES – Manuales:
  http://es.tldp.org/htmls/manuales.html

- Guía de Programación de Módulos del Núcleo Linux: http://www.ibiblio.org/pub/linux/docs/LuCaS/Manuales-LuCAS/doc-progmodlinux/doc-progmodlinux-html/progmodlinux.html

- How to configure NFS on Linux: http://linuxconfig.org/how-to-configure-nfs-on-linux

- Gestión de Entrada/Salida: http://www.ditec.um.es/so/apuntes/teoria/tema6.pdf

- Llamadas a procedimientos remotos: http://www.arcos.inf.uc3m.es/~infosd/lib/exe/fetch.php?media=es:t6_llamadas_a_procedimientos_remotos.pdf

- Linux Cross Reference: http://lxr.free-electrons.com/ident?i=register_chrdev

- Dispositivos y manejadores de dispositivos: http://sistemasoperativos.angelfire.com/html/4.1.html

- Sistemas operativos: http://oskarj023.blogspot.com.es/2012_08_01_archive.html

- Información generalizada: http://es.wikipedia.org

- Llamada a procedimientos remotos (RPC): http://ccia.ei.uvigo.es/docencia/SCS/1011/transparencias/Tema2-2.pdf

- Introducción a programación de módulos de kernel Linux: http://wiki.inf.utfsm.cl/index.php?title=Introducci%C3%B3n_a_programaci%C3%B3n_de_m%C3%B3dulos_de_kernel_Linux

- CARGAR MÓDULOS: http://sopa.dis.ulpgc.es/ii-aso/portal_aso/practicas/modulo.htm

- Módulos del núcleo: http://web-sisop.disca.upv.es/gii-dso/es/pl2-modulos/pl2-modulos.html

- Linux kernel map in printable PDF: http://www.makelinux.net/ldd3/chp-3-sect-4

- register_chrdev:
  https://www.fsl.cs.sunysb.edu/kernel-api/re941.html

- The linux-kernel mailing list FAQ:
  http://www.tux.org/lkml/

- Compartición de archivos en Linux mediante NFS:
  http://www.elrincondelprogramador.com

- Manejadores de dispositivos:
  http://web-sisop.disca.upv.es/gii-dso/es/pl3-dispositivo/pl3-dispositivo.html

- Sistemas de Entrada/Salida
  http://ecaths1.s3.amazonaws.com/funcionesdesistemaoperativoub/Sistemas%20de%20Entrada%20Salida.pdf

- El Sistema Ficheros en Red (NFS):
  http://www.tldp.org/pub/Linux/docs/ldp-archived/network-guide/translations/es/garl-1.0/garl11.htm

- Introducción al Network Information Service y Network File System:
  https://www.google.es/url?sa=t&rct=j&q=&esrc=s&source=web&cd=3&cad=rja&uact=8&ved=0CCoQFjACahUKEwigxr2siN3IAhVLlxoKHbAlAKQ&url=http%3A%2F%2Fseaceptanideas.com%2Fpdf_server.php%3Ffile%3D.%2Fcursos%2FIFCAD460%2FIFCAD460.pdf&usg=AFQjCNFOZk5CXQ-BT6DseKMkNjomkpufKQ&sig2=yXgfs2rqQixJuZxoF0-quA&bvm=bv.105841590,d.d2s

- Sistemas Operativos Distribuidos:
  https://www.google.es/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&cad=rja&uact=8&ved=0CCcQFjABahUKEwjHlbHTid3IAhVLORoKHRZ5Ahg&url=http%3A%2F%2Fwww.cs.buap.mx%2F~hilda%2Fnotasdistribuidos.doc&usg=AFQjCNGq-ee8C8D4X5Yhf5vbBXboC4HjTA&sig2=DcxXqDioZJTmmVXSaL-8Tg

- Sistemas de archivos distribuidos:
  http://sistemadearchivosdistribuidos03.blogspot.com.es/2015/06/que-es-un-sistema-de-archivos_29.html

- Sistemas de ficheros distribuidos:
  http://www.sc.ehu.es/acwlaroa/SDI/Apuntes/Cap4.pdf

- Sistemas Operativos Distribuidos:
  http://laurel.datsi.fi.upm.es/_media/docencia/asignaturas/sod/sfd_4pp.pdf

- PDF´s y PPT´s:
  - PDF “Drivers en Linux” del profesor Mario Muñoz.
  - Syscall: Llamadas al sistema Diseño de Sistemas Operativos.
  - DRIVERS LINUX.
  - Kernel y módulos.
  - Sistemas de Archivos Distribuidos.